WBEM on Sun Developer's Guide

Sun Microsystems, Inc.
901 San Antonio Road
Palo Alto, CA 94303
U.S.A.

Part No: 805-7995–10
August, 1999

Contents

Tables

Figures

Code Examples

Preface

This guide describes the Sun WBEM SDK and Solaris WBEM Services. The Sun WBEM SDK is a software developer's tool kit that software developers can use to create standards-based applications that manage WBEM-enabled objects. Developers can also use this software to write providers, programs that communicate with managed objects to access data.

Solaris WBEM Services software is Sun's implementation of WBEM on the Solaris™ operating environment. This software includes the Sun WBEM SDK and the core components needed to write WBEM-enabled management applications.

Who Should Use This Book

Two types of developers will use this book:

- System and Network Application Developers

  Programmers who write applications that manage the information stored in CIM classes and instances will find this guide useful. These developers typically use the Sun WBEM APIs to get and set the properties of predefined CIM instances and classes.

- Instrumentation Engineers

  Instrumentation engineers provide resources such as processors, memory, routers, and other manageable devices. These developers need to communicate device information in a standard CIM format to the CIM Object Manager, typically through a piece of software called a provider. These users use the WBEM APIs to create classes, instances, and properties.

  Instrumentation engineers might work with class and schema designers, who describe new groups of managed resources (CIM classes) or a collection of CIM

classes, called schema. Schemas describe the managed objects in a particular system environment, for example, Microsoft Windows 32 or the Solaris operating environment.

Before You Read This Book

This book describes how to use the Sun WBEM applications and tools to write management applications.

This book requires knowledge of the following:

- Object-oriented programming concepts
- Java programming
- A solid understanding of Common Information Model (CIM) concepts.

If you are unfamiliar with these areas, you might find the following references useful:

- *Java™ How to Program*

  H. M. Deitel and P. J. Deitel, Prentice Hall, ISBN 0–13–263401–5

- *The Java Class Libraries, Second Edition, Volume 1*, Patrick Chan, Rosanna Lee, Douglas Kramer, Addison-Wesley, ISBN 0–201–31002–3
- *CIM Tutorial*, provided by the Distributed Management Task Force

The following Web sites are useful resources when working with WBEM technologies.

- Distributed Management Task Force (DMTF)

  See this site at `www.dmtf.org` for the latest developments on CIM, information about various working groups, and contact information for extending the CIM Schema.

- Rational Software

  See this site at `www.rational.com/uml` for documentation on the Unified Modeling Language (UML) and the Rose CASE tool.

How This Book is Organized

Part I introduces the Sun WBEM SDK and Solaris WBEM Services products.

Chapter 1 introduces Web-Based Enterprise Management (WBEM), the Common Information Model (CIM), Sun WBEM SDK, and Solaris WBEM Services.

Part II explains how to install and use the components in the Sun WBEM SDK.

Chapter 2 describes how to install the Sun WBEM SDK using the `pkgadd` command and how to remove the Sun WBEM SDK using the `pkgrm` command.

Chapter 3 describes the command syntax for the `mofc` command and how to compile a `.mof` file.

Chapter 4 describes how to use CIM WorkShop to manipulate CIM classes, instances, methods, and properties.

Chapter 5 provides an overview of the client APIs and examples of how to use them to create and manipulate CIM objects.

Chapter 6 explains how to use the Client APIs to write client applications.

Chapter 7 provides an overview of the provider APIs and explains how to write a provider, classes that mediate between managed objects and the CIM Object Manager.

Chapter 8 explains how to run the code examples provided with the Sun WBEM SDK.

Chapter 9 explains error messages returned by Sun WBEM SDK APIs.

Part III explains how to install and use Solaris WBEM Services, Sun's implementation of WBEM on the Solaris operating environment.

Chapter 10 describes how to install Solaris WBEM Services using the `pkgadd` command and how to remove the software using the `pkgrm` command.

Chapter 11 describes the command syntax of the `cimom` command, and how to start and stop the CIM Object Manager.

Chapter 12 describes security features and how to set access rights on namespaces and users.

Chapter 13 describes the logging features.

Appendix A describes CIM terms and concepts.

Glossary presents a list of words and phrases found in this book and their definitions.

Ordering Sun Documents

The SunStore[SM] stocks hundreds of manuals from Sun Microsystems, Inc. You can purchase both documentation sets and individual manuals.

For a list of documents and how to order them, visit the SunStore at `http://sunstore.sun.com`.

Accessing Sun Documentation Online

You can access Sun technical documentation online at the docs.sun.com[SM] Web site. You can browse the archive or search for a specific book title or subject.

What Typographic Conventions Mean

The following table describes the typographic changes used in this book.

TABLE P–1 Typographic Conventions

Typeface or Symbol	**Meaning**	**Example**
`AaBbCc123`	The names of commands, files, and directories; on-screen computer output	Edit your `.login` file. Use `ls –a` to list all files. `machine_name% you have mail.`
`AaBbCc123`	What you type, contrasted with on-screen computer output	`machine_name%` **`su`** `Password:`

TABLE P–1 Typographic Conventions *(continued)*

Typeface or Symbol	Meaning	Example
AaBbCc123	Command-line placeholder: replace with a real name or value	To delete a file, type **rm** *filename*.
AaBbCc123	Book titles, new words, or terms, or words to be emphasized.	Read Chapter 6 in *User's Guide*. These are called *class* options. You must be *root* to do this.

Shell Prompts in Command Examples

The following table shows the default system prompt and superuser prompt for the C shell, Bourne shell, and Korn shell.

TABLE P–2 Shell Prompts

Shell	Prompt
C shell prompt	`machine_name%`
C shell superuser prompt	`machine_name#`
Bourne shell and Korn shell prompt	`$`
Borne shell and Korn shell superuser prompt	#

PART I Introduction

The Sun WBEM SDK is a software developer's tool kit that includes components needed to write management applications that extend the Common Information Model and run in any Java environment. Developers can also use this tool kit to write providers, programs that communicate with managed objects to access data.

The Solaris WBEM Services is a software product that provides management and security services.

CHAPTER 1

Overview of WBEM

This chapter provides a detailed description of Web-Based Enterprise Management (WBEM) . The following topics are covered.

- About WBEM
- Common Information Model
- Managed Object Format
- CIM and Solaris

About WBEM

Web-Based Enterprise Management (WBEM) is an initiative and a technology. As an initiative, WBEM includes standards for managing systems, networks, users, and applications by using Internet technology. As a technology, WBEM provides a way for management applications to share management data independently of vendor, protocol, operating system, or management standard. By developing management applications according to WBEM principles, vendors can develop products that work together easily for low cost of development.

The Distributed Management Task Force (DMTF), a group representing corporations in the computer and telecommunications industries, is leading the effort to develop and disseminate standards for management of desktop environments, enterprise-wide systems, and the Internet. The goal of the DMTF is to develop an integrated approach to managing networks across platforms and protocols, resulting in cost-effective products that interoperate as flawlessly as possible. For information about DMTF initiatives and outcomes, see the DMTF web site at `www.dmtf.org`.

Common Information Model

The Common Information Model (CIM) is an approach to managing systems and networks. CIM provides a common conceptual framework to classify and define the parts of a networked environment and depict how they integrate. The model captures notions that are applicable to all areas of management, independent of technology implementation.

CIM Terminology

The Common Information Model uses a set of terminology specific to the model and the principles of object-oriented programming. For information about CIM terminology and descriptions of what the terms represent, see Appendix A. See the *Glossary* for an expanded list of terms that have a specialized meaning in CIM.

CIM Structure

The Common Information Model categorizes information from general to specific. Specific information, such as a representation of the Solaris environment, extends the model. CIM consists of the following three layers of information:

- Core Model – A subset of CIM not specific to any platform.
- Common Model – Information model that visually depicts concepts, functionality, and representations of entities related to specific areas of network management, such as systems, devices, and applications.
- Extensions – Information models that support the CIM Schema and represent a very specific platform, protocol, or corporate brand.

Collectively, the Core Model and the Common Model are referred to as the CIM Schema.

The Core Model

The Core Model provides the underlying, general assumptions of the managed environment—for example, that specific, requested data must be contained in a location and distributed to requesting applications or users. These assumptions are conveyed as a set of classes and associations that conceptually form the basis of the managed environment. The Core Model is meant to introduce uniformity across schemas intended to represent specific aspects of the managed environment.

For applications developers, the Core Model provides a set of classes, associations, and properties that can be used as a starting point to describe managed systems and determine how to extend the Common Model. The Core Model establishes a conceptual framework for modeling the rest of the managed environment.

The Core Model provides classes and associations to extend specific information about systems, applications, networks, devices, and other network features through the Common Model and extensions. For information about the system aspects of the Core Model and related classes and associations, see “Core Model Concepts” on page 179.

The Common Model

Areas of network management depicted in the Common Model are independent of a specific technology or implementation but provide the basis for the development of management applications. This model provides a set of base classes for extension into the area of five designated technology-specific schemas: Systems, Devices, Applications, Networks, and Physical. For information about each of these schemas, see “Common Model Schemas” on page 183.

CIM Extensions

Extension schemas are built into CIM to connect specific technologies into the model. By extending CIM, a specific operating environment such as Solaris can be made available to a greater number of users and administrators. Extension schemas provide classes for software developers to build applications that manage and administer the extended technology.

Managed Object Format

MOF is the standard language used to define elements of the Common Information Model (CIM). The MOF language specifies a syntax for defining CIM classes and instances. MOF provides developers and administrators with a simple and fast technique for modifying the CIM Repository. For more information about MOF, see the DMTF web page at `http://www.dmtf.org`.

Because MOF can be converted to Java, an application developed in MOF can be run on any system or in any environment that supports Java.

The MOF Syntax

Programmers can use the CIM API to represent CIM objects, developed in MOF, as Java classes. The CIM Object Manager checks and enforces that these CIM objects comply with the CIM 2.1 Specification. In some cases, it is possible to represent something syntactically correct in a MOF file that does not adhere to the CIM specification. The CIM Object Manager returns an error message when such a MOF file is compiled.

For example, if you specify scope in the qualifier definition in a MOF file, CIM Object Manager returns a compilation error because scope can only be specified in the definition of a CIM Qualifier Type. A CIM Qualifier cannot change the scope that was specified in the CIM Qualifier Type.

Schema MOF Files

The installation of Solaris WBEM Services puts MOF files that form the CIM Schema and the Solaris Schema in the directory `/opt/SUNWconn/wbem/schema`. These files are automatically compiled and run when the CIM Object Manager starts.

The CIM Schema files, denoted by CIM in the file name, form standard CIM objects. For an explanation of the parts that make up the CIM Schema, see version 2.1 of the CIM Specification, which can be obtained at `http://dmtf.org/spec/cims.html`.

The Solaris Schema describes Solaris objects by extending the standard CIM Schema. The MOF files that make up the Solaris Schema use the Solaris prefix in the file names, but otherwise follow the same file name conventions as the CIM Schema MOF files. You can view the MOF files that make up the Solaris Schema in a text editor of your choice.

CIM and Solaris

Sun Microsystems, Inc. extends CIM principles and classes in the Solaris operating environment. Developed in Java to run on any Java-enabled platform, the Sun implementation consists of two products: the Sun WBEM SDK and Solaris WBEM Services.

Sun WBEM SDK

The Sun WBEM Software Development Kit (SDK) contains the components required to write management applications that can communicate with any WBEM-enabled management device. Developers can also use this tool kit to write providers,

programs that communicate with managed objects to access data. All management applications developed using Sun WBEM SDK run on the Java platform. For more information about Sun WBEM SDK components and installation instructions, see Chapter 2.

Solaris WBEM Services

Solaris WBEM Services provides routing and security services. The CIM Object Manager routes data about objects and events between components. Sun WBEM User Manager is an application with a graphical user interface in which you can set user permissions to specific work areas. For more information about Solaris WBEM Services components and installation instructions, see Chapter 10.

PART II Sun WBEM SDK

The Sun WBEM SDK is a software developer's tool kit that includes components needed to write Java management applications that can communicate with any WBEM-enabled management device. Developers can also use this tool kit to write providers, programs that communicate with managed objects to access data.

CHAPTER 2

Installing the Sun WBEM SDK

This section describes the Sun WBEM SDK and explains how to install and remove it from your system. Topics covered include the following:

- About the Sun WBEM SDK
- Installation Prerequisites
- Installing the Sun WBEM SDK
- Getting Started with the Sun WBEM SDK
- Removing the Sun WBEM SDK

About the Sun WBEM SDK

Sun WBEM SDK is a software developer's tool kit you can use to develop applications that can run in WBEM environments. You can also develop providers, programs that communicate between WBEM components and the CIM Object Manager, to run on the Java platform in any WBEM-enabled environment.

Providers must be started on a computer running the CIM Object Manager. For Sun WBEM SDK version 1.0, the CIM Object Manager is not supported in the Windows environment. Consequently, the Sun WBEM SDK is not recommended for the development of providers for the Microsoft Windows 32–bit operating system.

Sun WBEM SDK includes the following components:

- CIM WorkShop
- Client API
- Provider API
- MOF Compiler

- Sample client programs
- Sample provider programs
- User documentation
  - This guide
  - *Javadoc for Client and Provider API*

CIM WorkShop

CIM WorkShop is a software application in which you can view CIM classes and instances, create new CIM subclasses and instances, and add or delete new properties, methods, and qualifiers of subclasses. For information about how to use CIM WorkShop, see Chapter 4.

Client API

The Client Application Programming Interface (API) is a public API that Java applications use to request operations from the CIM Object Manager. For a complete list and description of the Client API, see Chapter 5.

Provider API

Provider APIs are interfaces that the CIM Object Manager and object providers use to communicate with each other. Providers can use these interfaces to provide the CIM Object Manager a particular kind of dynamic data. When an application requests dynamic data from the CIM Object Manager, the CIM Object Manager uses these interfaces to pass the request to the provider that registered to be the provider. For a description of the Provider API, see Chapter 5.

MOF Compiler

The Managed Object Format (MOF) is a standard text format for representing CIM data. Products developed according to the CIM specification must be able to exchange information in MOF format.

The Sun WBEM SDK software uses a MOF Compiler (`mofc`) to convert a MOF text file to Java classes. Internally, Sun WBEM SDK uses Java classes to represent CIM data.

The MOF language defines a syntax for defining CIM classes and instances. MOF provides developers and administrators with a simple and fast technique for modifying the CIM Object Manager Repository.

Sample Client Programs

The Sun WBEM SDK provides sample Java programs that use the Client APIs. These programs are installed in `/opt/SUNWconn/wbem/demo`. You can run the sample client programs and use them to build your own applications. For information about the sample client programs, see “Using Client Examples” on page 102 in the chapter, “Using Sun WBEM SDK Examples.”

Sample Provider Programs

Sample provider programs use the Provider APIs. You can use these programs to develop providers that extend CIM capabilities into your specific technology. For example, you can develop a provider that enables your technology to communicate with the CIM Object Manager. For information about the sample provider programs, see “Using the Provider Examples” on page 105 in the chapter, “Using Sun WBEM SDK Examples.”

Documentation

This guide and the Javadoc reference pages are provided as part of Sun WBEM SDK.

Installation Prerequisites

Prior to installing the Sun WBEM SDK kits, the Java Development Kit (JDK) version 1.1.7_05 or a compatible version must be installed on the server designated to run the CIM Object Manager.

Shared Packages

You can install Sun WBEM SDK as a product that runs on its own, or you can install both Sun WBEM SDK and Solaris WBEM Services to be used interactively. Installing

either product involves installing the product packages. The packages are compilations of the files, interfaces, and components of each product.

Sun WBEM SDK and Solaris WBEM Services share some of the same packages. For example, both applications require the package named `SUNWwbapi`, that contains the Client APIs.

For information about Sun WBEM SDK packages and installation instructions, see the following section, “Installing the Sun WBEM SDK” on page 14. For information about Solaris WBEM Services packages and installation instructions, see the following section, “Installing Solaris WBEM Services” on page 140 in Chapter 10.

Installing the Sun WBEM SDK

The following table describes the packages you need to install Sun WBEM SDK.

TABLE 2–1 Sun WBEM SDK Packages

Required Packages		
Package Name	**Title**	**Description**
`SUNWwbapi`	Sun WBEM SDK - APIs	Contains the client and provider APIs and additional functionality required to run the Sun WBEM SDK and Solaris WBEM Services. This package is provided with the Sun WBEM SDK. It is required by both products.
`SUNWwbdev`	Sun WBEM Software Development Kit (SDK)	Contains the MOF Compiler, CIM Workshop, context help used in CIM Workshop, and graphics files that make up the Sun WBEM SDK.
Optional Packages		
Package Name	**Title**	**Description**
`SUNWwbdoc`	Solaris WBEM Services - Documentation	Contains this guide, which supports both the Sun WBEM SDK and Solaris WBEM Services. Although this package is provided with Solaris WBEM Services, it can be installed optionally to support either product.
Localized Packages		

TABLE 2–1 Sun WBEM SDK Packages *(continued)*

Package Name	Title	Description
SUNW*xx*wbd	Sun WBEM SDK - Localization	Contains the localized version of the Solaris WBEM Services. The *xx* is replaced by the character code that represents the particular language in which the application is localized. For example, the French version of the Sun WBEM SDK is packaged in SUNWfrwbd.

▼ How to Install Sun WBEM SDK

1. **Become root on your system by typing the following command:**

```
% su
```

2. **Type the root password when you are prompted.**

3. **Change directories to the location of the packages in your work environment.**

4. **At the system prompt, type the following command to obtain a list of packages:**

```
# pkgadd -d .
```

   The list of packages is displayed. You are prompted to select one or all packages.

5. **Type the number of the package you want to install.**
   - Type 1 to install the SUNWwbapi package. It is important to install this package first because the other packages rely on the Sun WBEM APIs.
   - Type 5 to install the SUNWwbdev package, which installs the Sun WBEM SDK.
   - (Optional): Type 3 to install the SUNWwbdoc package, which installs this guide.

   As each package installs, its contents are listed for you to view. When the installation is complete, you are notified with the message:
   `Installation of package_name was successful.`

6. **When you have finished installing the packages, type `q` to exit the package installation routine.**

7. **Type `exit` at the system prompt to exit root.**

Getting Started with Sun WBEM SDK

After you finish installing the Sun WBEM SDK, you can use the product components. Getting started entails completing tasks that you will use on a daily basis, including the following:

- "Starting CIM WorkShop" on page 24
- "Compiling a MOF File" on page 21
- "Restarting the CIM Object Manager" on page 150

Uninstalling the Sun WBEM SDK

When you want to uninstall the Sun WBEM SDK from your computer, you remove the packages. When you remove the Sun WBEM SDK packages, not all files that make up your total installation are removed. If Solaris WBEM Services is installed, none of its associated packages are removed. For information about removing Solaris WBEM Services, see Chapter 10.

After you remove both the Sun WBEM SDK and Solaris WBEM Services, the LDAP schema and data files remain installed. You can remove these files, and the subdirectories that contain them, from the path `/opt/SUNWconn/ldap`. However, if you remove the LDAP data, you may encounter errors in other applications that require the data. Also, if you remove the LDAP data, you will need to re-install it if you decide to re-install the Sun WBEM SDK or Solaris WBEM Services at a later date.

▼ How to Uninstall the Sun WBEM SDK

1. **Become root on your system by typing the following command:**

```
% su
```

2. **Type the root password at the Password prompt.**

3. **Type the following command at the system prompt to remove a package:**

```
# pkgrm package_name
```

   where package_name is replaced by the name of the package that you want to remove.

4. **Type** `y` **when you are prompted with the question:**
   `"Do you want to remove this package?"`

   You can remove the following packages in any order:

   - `SUNWwbdev`
   - `SUNWwbdoc`

   Be sure to remove the `SUNWwbapi` package last because all other packages rely on it.

   When a package has been removed successfully, the following message is displayed.

```
Removal of package_name was successful
```

5. **Type the** `pkgrm` **command at the system prompt for each package you want to remove.**

6. **Type** `exit` **to exit root and return to your system prompt when you have finished removing packages.**

CHAPTER 3

MOF Compiler

This chapter describes the MOF Compiler and explains how to use it. The following topics are covered.

- About the MOF Compiler
- Compiling a MOF File

About the MOF Compiler

The MOF Compiler parses files created in Managed Object Format, converts files to Java classes, and stores the extracted classes and instances in the CIM Repository.

MOF Compiler Location

During installation, the MOF Compiler is provided in the `SUNWwbmof` package. After installation, the MOF Compiler is located in the following path: `/opt/SUNWconn/wbem/bin/`.

MOF Compiler Parameters

The `mofcomp` command uses the following command-line parameters.

Parameter	Description
`-help`	Causes a man page to display with information about the MOF compiler command and parameters
`-version`	Causes the build version of the MOF Compiler to display
`-v` or `-verbose`	Turns on verbose mode, which displays compiler messages as the file compiles
`-c cimom_host`	Enables you to specify the name of a host computer that is running the CIM Object Manager
`-u`	Enables you to specify a user name when you want to compile a file that is protected by user name and password authentication
`-p`	Enables you to provide a password when you want to compile a file that is protected by user name and password authentication

Syntax of the MOF Compiler

The MOF Compiler uses the following command syntax:

```
% mofcomp filename.mof
```

where `mofcomp` is the command to run the MOF Compiler and *filename*`.mof` is the name of a MOF file to be compiled.

Security Advisory for Using Passwords

If you run a command with the *-p* or the *-up* parameters, and you include a password, another user can run the `ps` command or the `history` command to find out your password.

Note - If you run a command that requires you to provide your password, immediately change your password after running the command.

CODE EXAMPLE 3–1 Example of Unsafe Syntax

The following examples show use of the `mofcomp` command with the *-p* parameter:

```
mofcomp -p Log8Rif
```

and the *-up* parameters:

```
mofcomp -up molly Log8Rif
```

Change your password immediately after running the `mofcomp` command with the option to specify a password.

Compiling a MOF File

Using the MOF Compiler, you can compile all MOF files with or without a `.mof` extension. You can start the compiler in any directory where you want to compile a file. All MOF files that make up the CIM and Solaris Schemas are located in the path: `/opt/SUNWconn/wbem/schema`.

▼ How to Compile a MOF File

1. **Change directories to the location of the MOF Compiler.**

```
% cd /opt/SUNWconn/wbem/bin/
```

2. **To run the MOF Compiler without parameters, type the following command:**

```
% mofcomp <filename>
```

At the prompt, type the path and name of the MOF file that you want to compile. For example, type: `/opt/SUNWconn/wbem/schema/Solaris_Schema1.0.mof`

The MOF file is compiled.

Example of MOF Output

The following example shows MOF output after compiling the file:

CODE EXAMPLE 3–2 Sample MOF Output

```
Initializing CIMValue
Parsing input file
Parsing input file
```

(Continuation)

```
Parsed input file
MofcBackend:
NamespaceTable:
{}End of NamespaceTable
QualifierTypesTable:
{}End of QualifierTypesTable
Syntax Errors: 0
Semantic Errors: 0
Warnings: 0
End of MofcBackend
```

CHAPTER 4

CIM WorkShop

This chapter explains how to use CIM WorkShop to add new properties, methods, and qualifiers to the classes and instances that you create, and to set the scope and flavor of new qualifiers for new classes and instances. The following topics are covered:

- About CIM WorkShop
- Starting CIM WorkShop
- Navigating in CIM WorkShop
- Viewing Class Characteristics
- Working in Namespaces
- Working with Classes
- Adding a Class
- Deleting Classes and Their Attributes
- Working with Instances
- Reference: CIM WorkShop Windows and Dialog Boxes

About CIM WorkShop

CIM WorkShop provides a graphical user interface in which you can view and create classes and instances. In CIM WorkShop, you can complete any of the following tasks:

- View and select namespaces
- Add namespaces
- View and create classes

- Add properties, qualifiers, and methods to new classes
- View and create instances
- View and modify instance values

Note - CIM guidelines prevent you from modifying or editing the properties, methods, or qualifiers of CIM Schema or Solaris Schema classes. However, you can create new classes and instances of classes. When you create a new class or instance, you can add or delete properties, methods, and qualifiers. You can also change the values, including the scope and flavor, of new qualifiers that you create for a new class, instance, property, or method. You cannot change the values of inherited properties, methods, or qualifiers.

Starting CIM WorkShop

CIM WorkShop is available as part of the Sun WBEM SDK.

The CIM Object Manager must be installed to run CIM WorkShop. During an installation of Solaris WBEM Services and Sun WBEM SDK in the Solaris operating environment, the CIM Object Manager runs on the local host. If you install only the Sun WBEM SDK, you must point to a host on which the CIM Object Manager already has started. You can enter this information in the Host field of the Login dialog box that is displayed when you start CIM WorkShop. For information about the CIM WorkShop dialog boxes and fields, see “Reference: CIM WorkShop Window and Dialogs” on page 38.

▼ How to Start CIM WorkShop

1. **Start the CIM WorkShop:**

   - If you have installed Solaris WBEM Services and Sun WBEM SDK in the Solaris operating environment, type the following command at the system prompt:

     ```
     % /opt/SUNWconn/wbem/bin/cimworkshop &
     ```

   - If you have installed the Sun WBEM SDK in the Microsoft Windows environment, click Start->Programs->WBEM SDK->CIM Workshop.

   The CIM WorkShop window is displayed, followed by the Login dialog box. The Login dialog box shows the name of the host computer on which CIM Workshop is installed and the path of the default namespace, `root\cimv2`. Context Help,

information about how to complete the dialog box, is displayed in the right side of the Login dialog box. When you click a field, the help content changes to reflect how to enter information into the field and the significance of the field.

2. **In the CIM WorkShop login dialog box, do the following:**

   - In the Host Name field, type the name of a host running the CIM Object Manager.

     Note - By default, CIM WorkShop connects to the CIM Object Manager on the local host, in the default namespace, `root\cimv2`. If you start CIM WorkShop as part of the WBEM SDK in the Solaris operating environment or in the Microsoft Windows environment, you need to provide the name of a host that is already running a CIM Object Manager.

   - In the Namespace field, click in the field and type the name of the namespace that you want to use, or retain the name of the default namespace.
   - In the User Name field, type the user name you generally use for system and networking privileges.
   - In the Password field, type the password you generally use for system and networking privileges.

     Note - If you do not specify a user name and password, you can log in using the default user account, *guest*. Guest privileges are read-only. Your CIM Object Manager administrator can set up write privileges associated with your user name and password.

     ---

3. **Click OK.**

   A message is displayed to show that the classes in the class inheritance tree are being enumerated. In the left side of the CIM WorkShop window, CIM classes are displayed.

Navigating in CIM WorkShop

When you first start CIM WorkShop, the classes of the CIM Schema display hierarchically in the left side of the CIM WorkShop window. This arrangement of classes is referred to as the class inheritance tree. When you select a class, its associated properties are listed in the right side of the window. In the following

illustration, the properties of the class `solaris_computersystem` are listed in the right side of the CIM WorkShop window.

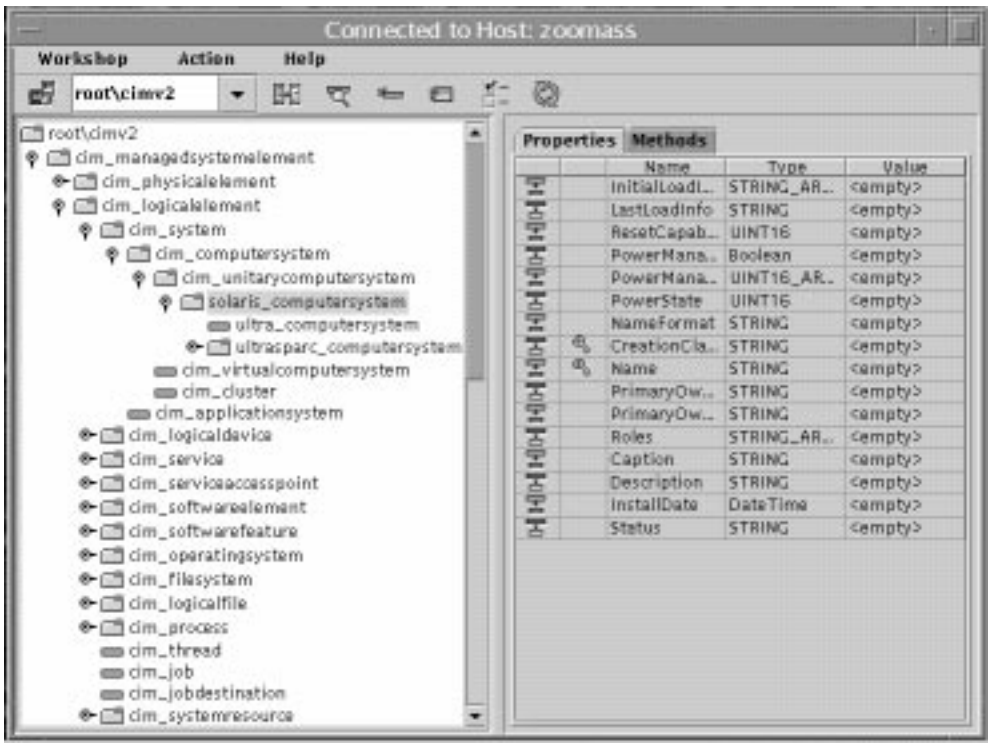

Figure 4–1 Class Inheritance Tree in CIM WorkShop Window

For information about the toolbar, menus, and layout of the CIM WorkShop window, see “Reference: CIM WorkShop Window and Dialogs” on page 38.

Browsing the Class Inheritance Tree

Each class that has classes is denoted by two icons: a folder icon and an enabler icon. The enabler icon looks like a small key to the left of the folder icon.

The folder icon indicates that the class serves as a container for the classes it contains. The enabler icon serves as a navigation aid.

When an enabler icon is displayed horizontally, the class folder is closed and classes are contained. When you click the enabler icon, the class folder opens and classes are revealed. When the enabler icon is displayed vertically, it indicates that the class folder is open.

▼ How To Display the Contents of a Class

- **Click the enabler icon of the desired class to view its contents.**

▼ How to Display the Properties and Methods of a Class

- **Click the class folder icon of the class.**

  The properties and methods of the class are displayed in the right frame of the CIM WorkShop window.

Finding a Class

CIM WorkShop enables you to quickly find a specific class.

▼ How to Find a Class

1. **In the toolbar, click the Find Class icon.**
2. **In the Find Class dialog box, type the name of the class you want to find and click OK.**

   When the specified class is found, its details are displayed in the right frame of the CIM WorkShop window.

Viewing Class Characteristics

When you select a class from the class inheritance tree—by clicking its folder icon—two tabs, indicating the properties and methods of the class, are displayed in the right side of the CIM WorkShop window.

Selecting a Class

In the class inheritance tree, classes that contain classes are indicated by folder icons. Classes that do not contain classes are indicated by purple rectangles. Select a class by clicking the folder or page icon of the class in the class inheritance tree.

Viewing Class Properties

By default, when the CIM WorkShop window is displayed, the Properties tab appears in the right side of the CIM WorkShop window. In the left side of the CIM WorkShop window, you can select a class from the class inheritance tree; you can view all properties of the class in the Properties tab. Inherited properties are indicated by an icon that consists of a purple rectangle with a black arrow pointing to a white rectangle. Properties that have an assigned key qualifier are indicated by a gold key icon. For information about the presentation of properties in the Properties tab, see “The Properties Tab” on page 41.

Viewing Class Methods

After you select a class in the class inheritance tree, you can click the Methods tab to display the methods associated with the class. For information about how the methods are displayed in the methods tab, see “The Methods Tab” on page 41.

Viewing Qualifiers

In CIM, qualifiers are attributes of classes, instances, properties, and methods. In CIM Workshop, you can view the qualifiers by right-clicking a class, property, or method and clicking Qualifiers in the pop-up menu. Clicking Qualifiers causes the Qualifiers dialog box to be displayed. For information about the presentation of qualifier information in the Qualifiers dialog box, see “Qualifiers Dialog Box” on page 44.

Viewing the Scope of a Qualifier

When you click the Scope button in the Qualifiers dialog box, the Scope dialog box is displayed. In the Scope dialog box, you can view the scope of a qualifier. When you create qualifiers for a new class, you can also set the scope of new qualifiers in the Scope dialog box. For information about the Scope dialog box, see “Scope Dialog Box” on page 45.

Viewing the Flavor of a Qualifier

When you click the Flavor button in the Qualifiers dialog box, the Flavors dialog box is displayed. In the Flavors dialog box, you can view the flavor of a qualifier. When you create qualifiers for a new class, you can also set the flavor of new qualifiers in the Flavors dialog box. For information about the Flavors dialog box, see “Flavors Dialog Box” on page 46.

Working in Namespaces

A namespace is a logical entity, an abstraction of a managed object into which classes and instances can be stored. A namespace can be implemented in various forms including a directory structure, a database, or a folder. By default, CIM WorkShop connects to the CIM Object Manager on the local host, in the default namespace `root\cimv2`. All classes contained in the default namespace are displayed in the left side of the CIM WorkShop window. The name of the current namespace is listed in the toolbar of the CIM WorkShop window. In CIM WorkShop, you can browse the classes of namespaces on different hosts and you can change location to new namespaces.

When you want to set user privileges for a particular namespace, use the Sun WBEM User Manager. For information about the Sun WBEM User Manager tool, see “Using the Sun WBEM User Manager to Set Access Control” on page 155 in Chapter 12.

This section describes how to:

- Change to a namespace
- Change to a host
- Refresh the class inheritance tree of the namespace

Changing Namespaces

In the Sun WBEM SDK, the default namespace is `root\cimv2`. You can change to any other namespace.

▼ How to Change Namespaces

1. **In the CIM WorkShop window, click Workshop->Change Namespace.**

2. **In the Change Namespace dialog box, click the icon of the namespace you want to use. Click OK.**

   The namespace you have selected becomes the current namespace.

Changing Hosts

You can change to another host to view namespaces or processes.

▼ How to Change Hosts

1. **Click Workshop->Change Host or click the Change Hosts icon in the CIM WorkShop toolbar.**
2. **In the Hosts field, type the name of the host on which the namespace you want to view is located.**
3. **Type your user name and password in the User Name and Password fields, respectively.**
4. **Click OK.**

Refreshing Classes and Namespaces

You can refresh the display of the class inheritance tree in the namespace to reflect current changes made by other users who work in the namespace.

▼ How to Refresh a Class Inheritance Tree

1. **In the class inheritance tree, click the folder of the class you want to refresh.**
2. **Click Action->Refresh or click the Refresh Selected Class icon in the CIM WorkShop toolbar.**

Working with Classes

Classes are the building blocks of applications. When you start CIM WorkShop, it becomes populated with the classes that make up the CIM and Solaris Schemas. These classes adhere to the Distributed Management Task Force standards. Their unique properties, methods, and qualifier values cannot be changed.

To set new values for an existing class, you can create a new instance or class. The CIM and Solaris Schema classes serve as templates. When you create a new instance or class, you produce a copy of the selected class in which you can add new properties, methods, and qualifier values. In this way, you build your own extensions into the CIM or Solaris Schemas.

Note - You cannot modify the values of inherited properties, methods, or qualifiers.

For information about how to create an instance, see “Working with Instances” on page 36. For information about how to create a class, see the following section.

Adding a Class

Adding a class to an existing class entails the following tasks:

- Selecting the class
- Creating a new class
- Adding new qualifiers to the class
- Adding new properties to the class
- Adding new qualifiers to the properties
- Adding new methods to the class
- Adding new qualifiers to the methods
- Setting qualifier values: scope and flavor

Creating a Class

The first step in creating a class is to specify a name for the class. In CIM WorkShop, class names are displayed using standard CIM syntax: *SchemaIndicator_ClassName*. If you create a class of a CIM Schema class, the acronym *CIM* is used before the class

name. If you create a class of a Solaris Schema class, the name *Solaris* is used before the class name. The underscore character (_) is required in the name of all classes that inherit a Key qualifier.

▼ How to Add a class

1. **In the class inheritance tree of the CIM WorkShop window, select the class from which to create a class.**

2. **Choose one of the following procedures for creating a class:**
   - Click Action->Add Class.

     or
   - Click the New Class icon in the toolbar of the CIM WorkShop window.

     or
   - Right-click the selected class and click Add Class.

     The New Class dialog box is displayed.

3. **In the Class Name field, type the name of the new class.**

   For example, you can create a class of the class `solaris_computersystem` titled `ultra1_computersystem`.

4. **To retain inherited properties and methods of the class, click OK. To add new properties, click Add Property.**

   If you click OK, a class is created that uses inherited properties, methods, qualifiers, and their values. If you click Add Property, the Add Properties dialog box is displayed, in which you can specify properties to add to the class. For information about how to add properties to a class, see "Adding New Properties to a Class" on page 33.

Adding Qualifiers

You can add qualifiers to a new class. You cannot change or reset the values of inherited qualifiers that modify the class. Also, you cannot delete inherited qualifiers.

▼ How to Add Qualifiers

1. **In the New Class dialog box, after you provide a name for the new class, click Class Qualifiers.**

2. **In the Qualifiers dialog box, right-click the Qualifier for which you want to set new values and click Add Qualifier.**

3. **In the Add Qualifier dialog box, select the name of a qualifier in the list and click OK.**

4. **To set the scope of the qualifier:**
   a. **Click Scope.**
   b. **In the Scope dialog box, select the scope of the qualifier and click OK.**

5. **To set the flavor of the qualifier:**
   a. **Click Flavors.**
   b. **In the Flavors dialog box, select the flavor of the qualifier and click OK.**

6. **Click OK in the Qualifiers dialog box to close it.**

Adding New Properties to a Class

You can add new properties to a class and modify their values. You cannot change the values of inherited properties, and you cannot delete inherited properties.

▼ How to Add a New Property to a Class

1. **After specifying a name for the new class, click Add Property in the New Class dialog box.**
   The Add Properties dialog box is displayed.

2. **In the Name field, type the name of the new property.**

3. **Select a property type from the Type field and click OK.**
   The new property is displayed in the Properties tab of the New Class dialog box. If the list of properties is long, click the scroll bar to view the newly added property.

4. **Click OK in the New Class dialog box.**
   For information about how to add new qualifiers or set qualifier values for a new property or class, see the following sections.

Adding Qualifiers to a New Property

You can set the values of qualifiers for new properties of the class. You cannot change or reset the values of qualifiers that modify inherited properties or methods. You cannot delete inherited qualifiers.

▼ How to Add Qualifiers to a New Property

1. **In the New Class dialog box, click the new property you created and click Property Qualifiers.**
   The Qualifiers dialog box is displayed for the property that you created.
2. **Click Add Qualifier.**
3. **In the Name field of the Add Qualifier dialog box, select a qualifier and click OK.**
4. **Click OK in the Qualifiers dialog box and in the New Class dialog box.**
   The qualifier and qualifier type are set for the selected property.

Deleting Classes and Their Attributes

CIM WorkShop provides a way to delete classes, properties, methods, and qualifiers that you no longer need or use.

Note - When you delete a class, you delete all subclasses it contains. You also delete all associated properties, methods, and qualifiers of the class and its subclasses.

Deleting a Class

Use the following procedure to delete a class from the class inheritance tree.

▼ How to Delete a Class

1. **Select the class that you want to delete.**

2. **Click Classes->Delete Class and click OK in the dialog box that asks you to confirm your decision to delete a class.**

   The class is deleted.

Deleting a Property of a Class

You can delete a property only when you create a new class. Otherwise, you can view but not modify or delete properties of classes. However, when you create a new class, you can delete properties inherited from the parent class if you do not need them or want to use them. For information about how to create a class, see “Adding a Class” on page 31.

▼ How to Delete a Property of a Class

1. **In the Properties tab of the New Class dialog box, right-click a property.**

2. **Click Delete Property in the pop-up menu.**

   The property is deleted from the Properties tab.

Deleting Qualifiers

When you create a new class, you can delete qualifiers of properties or methods inherited from the parent class. For information about how to create a class, see “Adding a Class” on page 31.

▼ How to Delete a Qualifier of a Property

1. **In the Properties tab of the New Class dialog box, right-click the property that you want to delete.**

2. **Click Qualifiers in the pop-up menu.**

3. **In the Qualifiers dialog box, click Delete Qualifier, then click OK.**

   The selected qualifier is deleted. The New Class dialog box is displayed.

▼ How to Delete a Qualifier of a Method

1. **In the Methods tab of the New Class dialog box, right-click the method that you want to delete.**
2. **Click Qualifiers in the pop-up menu.**
3. **In the Qualifiers dialog box, click Delete Qualifier, then click OK.**
   The selected qualifier is deleted. The New Class dialog box is displayed.

Working with Instances

In CIM WorkShop, you can create instances of classes. Instances inherit the characteristics of the class. You can then change the attributes of a new instance to create a unique instance of a class.

Displaying Instances

Before you create a new instance of a class, it is useful to view the instances of the class to see what properties and methods they contain.

▼ How to Display Instances of an Existing Class

1. **In the class inheritance tree of the CIM WorkShop window, select the class for which you want to view instances.**
2. **To display the Instances window:**
   - Click Action->Instances.

     or
   - Click the Show Instances icon on the CIM WorkShop toolbar.

The Instances window is displayed. If the selected class has instances, the instances are displayed in the left frame of the Instances window. If the selected class does not have instances, the left frame of the Instances window is empty.

Adding Instances

Add instances to a class when you want to modify the inherited qualities of objects.

▼ How to Add an Instance to a Class

1. **In the CIM WorkShop window:**
   - Click Action->Instances.

     or
   - Right-click a class that has instances from which you can create a new instance. Click Instances in the pop-up menu.

     The Instances window is displayed. All instances of the class are displayed in the left side of the window.
2. **Right-click an instance listed in the Instances window.**

   The Add Instances dialog box is displayed.
3. **To modify the inherited properties of an instance:**
   a. **Right-click a property listed in the Add Instances dialog box.**

      A dialog box is displayed in which you can provide a value for the property. The dialog box displayed varies depending on the type of the selected property. For example, if you select a property that has a type STRING, the String dialog box will display. The Value field of this dialog box accepts only character strings.

   b. **In the Values field of the dialog box, type the required value.**
4. **Click OK to close the Add Instances window.**

Deleting Instances

You can delete an instance that you no longer use.

▼ How to Delete an Instance

1. **In the left frame of the CIM WorkShop window, right-click the class from which you want to delete an instance.**

2. **In the pop-up menu, click Instance.**

3. **In the Instance window, right-click the instance you want to delete and click Delete Instance in the pop-up menu.**

   The instance is deleted.

Reference: CIM WorkShop Window and Dialogs

The following section provides descriptions of the frames, toolbar icons, and fields that comprise the CIM WorkShop window. It also describes CIM WorkShop dialogs.

The CIM WorkShop Window

The CIM WorkShop window is divided into two main frames. In the left frame, you can view the class inheritance tree of the current host. In the right frame, you can view the properties and methods of a selected class.

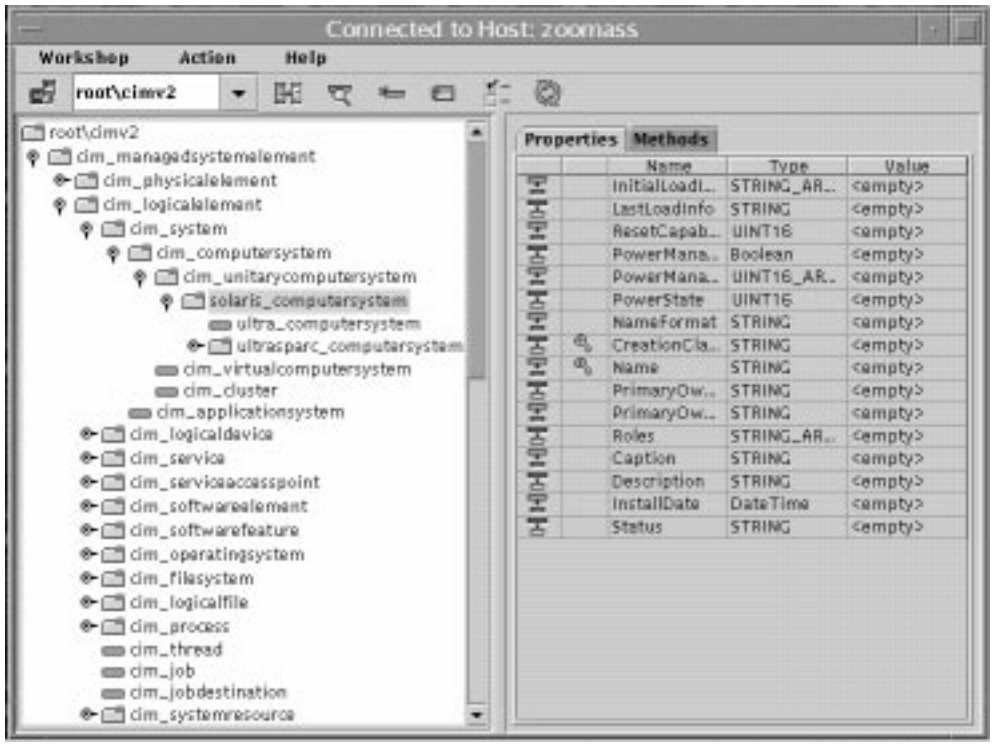

Figure 4–2 The CIM WorkShop Window

TABLE 4–1 Frames of the CIM WorkShop Window

Frame	Description
Left frame	Displays classes and instances contained in the namespace of the current host. The left frame in the CIM WorkShop shows the contents of the selected namespace. The classes that belong to the namespace are displayed hierarchically. This organization of classes is known as a class inheritance tree. Classes that contain subclasses are represented as a key icon and a folder. Clicking the key or double-clicking the folder causes the list of subclasses to display. Classes that do not contain subclasses are represented by page icons.
Right frame	Provides a Properties tab and a Methods tab from which you can view the values of properties and methods of a class. You can view attributes and values of qualifiers and flavors by right-clicking on a property or method.

TABLE 4–1 Frames of the CIM WorkShop Window *(continued)*

Frame	Description
Toolbar	Provides icons that enable you to change hosts, change location to a namespace within the default namespace /root/cimv2, find a class in the class inheritance tree, create a subclass, and show instances and qualifiers of a selected class.
Title bar	Posts the title of the CIM WorkShop window

CIM WorkShop Toolbar Icons

The icons provided in the CIM WorkShop toolbar enable you to display and change namespaces and search for classes and instances.

Figure 4–3 The CIM WorkShop Toolbar

TABLE 4–2 Icons of the CIM WorkShop Toolbar

Icon	Description
Change Hosts	Enables you to connect to a different host or name space and to log in with a different user name and password.
Change Namespace	Causes the Change Name Space dialog box to display in which you can select another name space to view.
Find Class	Enables you to search for a specific class in the name space.
New Class	Causes the New Class dialog box to display in which you can create a new subclass of a selected class.
Show Instances	Causes the Show Instances dialog box to display in which you can view instances of a selected class.

TABLE 4–2 Icons of the CIM WorkShop Toolbar *(continued)*

Icon	Description
Show Qualifiers	Causes the Qualifiers dialog box to display in which you can view the qualifiers of a selected class.
Refresh Selected Class	Resets the display of the class hierarchy tree. Open class folders are closed and the tree is returned to the state it was in when it was first displayed.

The Properties Tab

The Properties tab shows information about a selected property. An icon resembling a folder with an arrow indicates that the property is inherited from a superclass. An icon resembling a gold key indicates that the property is a Key. Key properties provide unique identifiers for an instance of the domain class. The unique instance is indicated by a key qualifier.

In the Properties tab, the Name and Type of the property are displayed. You can change the value of a property when you create a new class of the domain class.

The Methods Tab

By selecting the Methods tab, you can view all methods of the class. Methods are listed consecutively. Reading horizontally from left to right, the following attributes are displayed for each method:

TABLE 4–3 Method Tab Attributes

Method Attribute	Description	Example
Name	Name specified for the method	`GetDateTime`
Value	Value specified for the method, for example,	null
Type	Type of method	string

TABLE 4–3 Method Tab Attributes *(continued)*

Method Attribute	Description	Example
Qualifier flavor	Flavor of qualifier assigned to the method	TRANSLATABLE
Description	String that returns the current system date and time in CIM date-time format	19990519142015.0000000-300

CIM WorkShop Menus

The following table describes the CIM WorkShop menus and menu items.

TABLE 4–4 CIM WorkShop Menus and Menu Items

Menu	Menu Item	Description
Workshop	Change Host	Causes the Show Instances dialog box to display, in which you can view all instances of a selected class.
	Change Namespace	Causes the Change Namespace dialog box to display, in which you can change location to a namespace in the default /root/cimv2 namespace.
	Exit	Enables you to exit CIM Workshop.

TABLE 4–4 CIM WorkShop Menus and Menu Items *(continued)*

Menu	Menu Item	Description
Action	Add Class	Causes the New Class dialog box to display in which you can create a subclass for a selected class.
	Delete Class	Deletes a selected class.
	Find Class	Enables you to specify a class to find in the class inheritance tree.
	Instances	Causes the Instances dialog box to display for the selected class. In this dialog box, you can view all instances of the class, add new instances, and delete instances.
	Qualifiers	Causes the Qualifiers dialog box to display. In this dialog box, you can view qualifier values, scope, and flavor of a selected class.
	Refresh	Causes the latest changes to be retrieved from the CIM Object Manager and displayed in CIM WorkShop for a selected class or namespace.

Login Dialog Box

The login dialog box is displayed when you first encounter CIM WorkShop. In the login dialog box, you specify the following:

- Host on which the CIM Object Manager is running and which contains the namespace you want to use
- Namespace in which you want to work
- Your user name
- Your password

If you do not specify a user name and password, you log in to CIM WorkShop as a guest. Guest privileges are read-only.

New Class Dialog Box

In the New Class dialog box, you can create a new class.

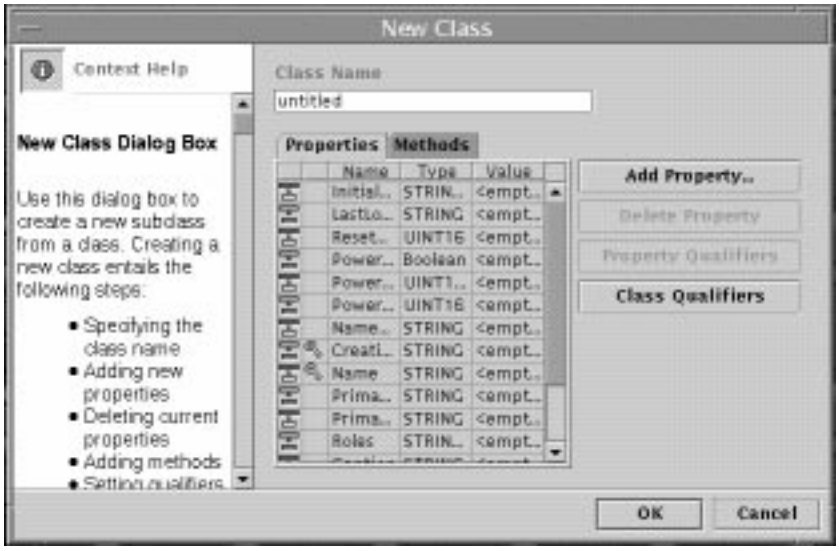

Figure 4–4 The New Class Dialog Box

Add Properties Dialog Box

In the Add Properties dialog box, you can add new properties to a class as you create it. In the Name field, you specify the name of the property. In the type field, select a type and click OK.

Qualifiers Dialog Box

In the Qualifiers dialog box, you can view qualifiers for a selected class, property, or method. When you create a new class, you can add qualifiers to the class or modify qualifiers of the class, its properties, or its methods in the Qualifiers dialog box. The title bar of the Qualifiers dialog box indicates the name of the class for which you view qualifiers or the class for which you add or modify qualifiers.

The following table describes the fields of the Qualifiers dialog box.

TABLE 4–5 Qualifiers Dialog Box Fields

Name of Field	Description	Example
Name	Shows the name of the qualifier	Provider
Type	Shows the type of value that the qualifier provides	`STRING`
Value	Shows the value of the qualifier	`Solaris`

The following table describes the buttons of the Qualifiers dialog box.

TABLE 4–6 Qualifiers Dialog Box Buttons

Name of Button	Description
Scope	Causes the Scope dialog box to display, in which you can view the scope of a selected qualifier
Flavors	Causes the Flavors dialog box to display, in which you can view the flavor of a selected qualifier
Add Qualifier	Causes the Add Qualifier dialog box to display, in which you can select a qualifier to add for a new subclass, property, or method
Delete Qualifier	Causes a selected qualifier to be deleted from the Qualifiers dialog box

Scope Dialog Box

In the Scope dialog box, you can view the scope of a qualifier that modifies an existing class, property, or method. You can also change the scope of a qualifier that you create for a new class or for a property or method added to a new class. The Scope field lists all possible scopes you can select.

Flavors Dialog Box

In the Flavors dialog box, you can view the flavor of a qualifier. You can also change the flavor of a qualifier that you create for a new class or for a property or method added to a new class. The Flavors field lists all possible flavors you can select.

Value Type Dialog Boxes

When you create new properties for a class, you can use any of the dialog boxes that CIM WorkShop provides for specifying properties of a particular type. These dialog boxes are configured to accept only a value of the appropriate type. The dialog boxes include the following:

- Real Integer dialog box
- Signed Integer dialog box
- Unsigned Integer dialog box
- String dialog box
- Array dialog box
- Boolean dialog box

Real Integer Dialog Box

The Values field of this dialog box accepts only a real integer. A real integer can be a negative or positive number including a decimal point. When you create a property that is of the type Real Integer, type a real integer in the Values field of this dialog box.

Signed Integer Dialog Box

The Values field of this dialog box accepts only a signed integer of a specified size. A signed integer can be a negative or positive whole number. CIM properties that have values which are signed integers can be 8 bits, 16 bits, 32 bits, or 64 bits in size. Depending on the size of the signed integer that makes up the value of the property, type the following in the Values field of this dialog box:

- For a property that is an 8–bit signed integer, type a positive or negative numeric value equivalent to 8 bits.
- For a property that is a 16–bit signed integer, type a positive or negative numeric value equivalent to 16 bits.
- For a property that is a 32–bit signed integer, type a positive or negative numeric value equivalent to 32 bits.

- For a property that is a 64–bit signed integer, type a positive or negative numeric value equivalent to 64 bits.

Unsigned Integer Dialog Box

The Values field of this dialog box accepts only an unsigned integer of a specified size. An unsigned integer can be only a positive whole number. CIM properties that have values which are unsigned integers can be 8 bits, 16 bits, 32 bits, or 64 bits in size. Depending on the size of the unsigned integer that makes up the value of the property, type the following in the Values field of this dialog box:

- For a property that is an 8–bit unsigned integer, type a positive numeric value equivalent to 8 bits.
- For a property that is a 16–bit unsigned integer, type a positive numeric value equivalent to 16 bits.
- For a property that is a 32–bit unsigned integer, type a positive numeric value equivalent to 32 bits.
- For a property that is a 64–bit unsigned integer, type a positive numeric value equivalent to 64 bits.

String Dialog Box

The Values field of this dialog box accepts only alphabetic and numeric characters. When you specify the value of a property that is a character string, you must enter a character string, such as Processor_Type, in the Values field of this dialog box. Character strings may not contain integers.

Array Dialog Boxes

In the Array dialog boxes, you can specify an array as a value for a property. The following array dialog boxes are available to return arrays:

- 8–Bit Unsigned Integer Array Dialog Box – returns a collection of positive integers equivalent to 8 bits in size
- 16–Bit Unsigned Integer Array Dialog Box – returns a collection of positive integers equivalent to 16 bits in size
- 32–Bit Unsigned Integer Array Dialog Box – returns a collection of positive integers equivalent to 32 bits in size
- 64–Bit Unisgned Integer Array Dialog Box – returns a collection of positive integers equivalent to 64 bits in size

- 8–Bit Signed Integer Array Dialog Box – returns a collection of positive or negative integers equivalent to 8 bits in size
- 16–Bit Signed Integer Array Dialog Box – returns a collection of positive or negative integers equivalent to 16 bits in size
- 32–Bit Signed Integer Array Dialog Box – returns a collection of positive or negative integers equivalent to 32 bits in size
- 64–Bit Signed Integer Array Dialog Box – returns a collection of positive or negative integers equivalent to 64 bits in size
- String Array Dialog Box – returns a collection of alphabetic and numeric character strings
- Boolean Array Dialog Box – returns a collection of Boolean expressions, True or False
- 32–Bit Real Array Dialog Box – returns a collection of positive or negative real numbers, with or without a decimal point, equivalent to 32 bits in size
- 64–Bit Real Array Dialog Box – returns a collection of positive or negative real numbers, with or without a decimal point, equivalent to 64 bits in size
- 16–Bit Character Array Dialog Box – returns a collection of alphabetic and numeric character strings equivalent to 16 bits in size
- Date/Time Array Dialog Box – returns a collection of dates in the format *mm-dd-yy* and times in the format *hh:mm:ss*

Boolean Dialog Box

In the Boolean dialog box, you can specify True or False as the value of a selected property.

Instance Window

The Instance window lists all instances for a selected class. You can also view the properties, methods, and qualifiers associated with each instance.

You display the Instance window by:

- Selecting a class in the CIM WorkShop window and clicking Action->Instances
- Right-clicking a class in the CIM WorkShop window and clicking Instances in the pop-up menu

Frames of the Instances Window

If the selected class has instances, the instances are listed in the left frame of the Instances window. Each instance is displayed with its Name, CreationClassName, and TargetOperatingSystem. If the selected class does not have instances, the left frame is empty.

Like the CIM WorkShop window, the right frame of the Instances window contains two tabs: Properties tab and Methods tab. All properties of the selected instance are displayed in the table of the Properties tab. The Inherited Properties icon—which appears in the left column of the Properties table as a purple rectangle with an arrow pointing to a white rectangle—indicates that the property is inherited from the class used to create the instance. The Key Qualifiers icon—which appears as a gold key—indicates that a property has an inherited Key qualifier.

Instances Window Toolbar Icons

The Instances window contains the following icons in the toolbar:

TABLE 4–7 Instance Window Toolbar Icons

Icon Name	Description
Add New Instance	Causes the Add Instance dialog box to display in which you can create a new instance to add to the class inheritance tree
Delete Selected Instance	Enables you to delete a selected instance
Refresh Instance List	Updates the list of instances in the left side of the Instances window with the newest instances and the latest changes to instances

Menus of the Instances Window

The instances window contains the following menus and menu items:

TABLE 4–8 Menus of the Instances Window

Menu Name	Menu Item	Description
Instance Editor	Exit	Causes the Instances window to close
Action	Add Instance	Causes the Add Instances dialog box to display, in which you can create new instances to add to the class inheritance tree
	Delete Instance	Enables you to delete a selected instance
	Refresh	Updates the list of instances in the left side of the Instances window with the newest instances and the latest changes to instances

Add Instances Dialog Box

In the Add Instances dialog box, you can right-click a property to change its value for a new instance. You cannot change the values of inherited properties.

CHAPTER 5

Application Programming Interfaces

The Sun WBEM SDK applications request information or services from the Common Information Model (CIM) Object Manager through the application programming interfaces (APIs). This chapter describes the following topics.

- About the APIs
- The API Packages

For detailed information on the CIM and Client APIs, see the Javadoc reference pages.

About the APIs

The APIs represent and manipulate CIM objects. These APIs represent CIM objects as Java classes. An object is a computer representation or model of a managed resource, such as a printer, disk drive, or CPU. Because the CIM Object Manager enforces the Common Information Model (CIM) 2.1 Specification, the objects you model using the APIs conform to standard CIM objects.

Programmers can use these interfaces to describe managed objects and retrieve information about managed objects in a particular system environment. The advantage of modeling managed resources using CIM is that those objects can be shared across any system that is CIM compliant.

The API Packages

The API can be grouped into three categories:

- CIM API – Common classes and methods that applications use to represent all basic CIM elements
- Client API – Methods that applications use to transfer data to and from the CIM Object Manager
- Provider API – Interfaces that the CIM Object Manager and object provider use to communicate with each other

CIM API Package (`com.sun.wbem.cim`)

The following table describes the interfaces in the CIM API package.

TABLE 5–1 CIM Classes

Class	Description
`CIMClass`	A CIM class, an object representing a collection of CIM instances, all of which support a common type (for example, a set of properties and methods). This interface creates a template that fills in the required CIM values for the group of objects you are creating.
`CIMDataType`	The CIM data types (as defined by the CIM specification).
`CIMDateTime`	The CIM date and time representation.
`CIMElement`	A CIM element. This is the base class for managed system elements.
`CIMFlavor`	A CIM qualifier flavor, a characteristic of a qualifier that describes rules that specify whether a qualifier can be propagated to derived classes and instances, and whether or not a derived class or instance can override the qualifier's original value.
`CIMInstance`	A unit of CIM data. Use this interface to describe a managed object that belongs to a particular class. Instances contain actual data.
`CIMMethod`	A declaration containing the method name, return type, and parameters.
`CIMNameSpace`	A CIM namespace, a directory-like structure, that can contain other namespaces, classes, instances, qualifier types, and qualifiers.

TABLE 5–1 CIM Classes *(continued)*

Class	Description
`CIMObjectPath`	A path name of a CIM object. The object names have two parts: namespace and a model path. The model path uniquely identifies an object in the namespace.
`CIMParameter`	A CIM parameter, a value passed to a CIM method from a calling method.
`CIMProperty`	A value that characterizes an instance of a CIM class. Properties can be thought of as a pair of functions, one to set the property value, and one to return the property value. A property has a name and one domain, the class that owns the property.
`CIMQualifier`	A modifier that describes a class, instance, method, or property. Use this class to modify an attribute of a managed object, for example, add read-only access to a disk. There are two categories of qualifiers: those defined by the Common Information Model (CIM) and those defined by developers.
`CIMQualifierType`	A CIM qualifier type, a template for creating CIM qualifiers.
`CIMQuery`	A CIM query, which specifies filters for indication subscription as well as for retrieval of CIM elements from the CIM Object Manager.
`CIMScope`	A CIM scope, a qualifier attribute that indicates the CIM objects with which the qualifier can be used. For example, the qualifier ABSTRACT has Scope(Class Association Indication), meaning that it can only be used with classes, associations, and indications.
`CIMValue`	A CIM value, a value that can be assigned to properties, references, and qualifiers. CIM values have a data type (`CIMDataType`) and the actual value(s).
`UnsignedInt8`	An unsigned 8-bit integer.
`UnsignedInt16`	An unsigned 16-bit integer.

TABLE 5–1 CIM Classes *(continued)*

Class	Description
`UnsignedInt32`	An unsigned 32-bit integer.
`UnsignedInt64`	An unsigned 64-bit integer.

Exception Classes

The Exception classes represent the error conditions that can occur in Sun WBEM SDK classes. The `CIMException` class is the base class for CIM exceptions. All other CIM exception classes extend from the `CIMException` class.

The following table describes the CIM exception classes.

TABLE 5–2 Exception Classes

Class	Description
`CIMClassException`	A semantic exception that occurs in a CIM class. The MOF Compiler (`mofc`) uses this class to handle semantic errors found during compilation.
`CIMException`	Exceptional CIM conditions. This is the base class for CIM exceptions.
`CIMInstanceException`	Semantic exceptions that occur in a CIM instance.
`CIMMethodException`	Semantic exceptions that occur in a CIM method.
`CIMNameSpaceException`	Semantic exceptions that occur in a CIM namespace.
`CIMPropertyException`	Semantic exceptions that occur in a CIM property.
`CIMProviderException`	Exceptional conditions that can occur in the CIM Object Manager's providers.

TABLE 5–2 Exception Classes *(continued)*

Class	Description
`CIMQualifierTypeException`	Exceptional conditions that can occur in a CIM qualifier type.
`CIMRepositoryException`	Exceptional conditions that can occur in the CIM repository.
`CIMSemanticException`	Semantic exceptions that can occur in a CIM element. These exceptions are generally thrown when the CIM Object Manager tries to add, modify, or delete a CIM element and encounters situations that are illegal according to the CIM Specification.
`CIMTransportException`	Exceptional conditions that occur in the CIM transport interfaces (RMI and XML).

Client API Package (`com.sun.wbem.client`)

The Client API package contains classes and methods that transfer data between applications and the CIM Object Manager. Applications use the `CIMClient` class to connect to the CIM Object Manager, and they use the methods listed in the following table in the `CIMClient` class to transfer data to and from the CIM Object Manager.

TABLE 5–3 Client Methods

Method	Description
`close`	Close the client connection to the CIM Object Manager. This interface frees resources used for the client session.
`createNameSpace`	Creates a CIM namespace, a directory containing classes and instances. When a client application connects to the CIM Object Manager, it specifies a namespace. All subsequent operations occur within that namespace on the CIM Object Manager host
`deleteNameSpace`	Deletes the specified namespace on the specified host.

TABLE 5–3 Client Methods *(continued)*

Method	Description
`deleteClass`	Deletes the specified class.
`deleteInstance`	Deletes the specified instance.
`deleteQualifierType`	Deletes the specified qualifier type.
`enumClass`	Retrieves the specified class or classes from the CIM Object Manager.
`enumInstances`	Gets a list of instances for the specified class or classes.
`enumNameSpace`	Gets a list of namespaces.
`enumQualifierTypes`	Gets a set of qualifier types for the specified class or classes.
`getClass`	Gets the CIM class for the specified CIM object path.
`getInstance`	Gets the CIM instance for the specified CIM object path.
`getQualifierType`	Gets the qualifier type for the specified CIM object path.
`invokeMethod`	Executes the specified method on the specified object. A method is a declaration containing the method name, return type, and parameters in the method.
`setClass`	Invokes the CIM Object Manager on this client to add the specified CIM class to the specified namespace.
`setInstance`	Invokes the CIM Object Manager to add or update the specified CIM instance to the specified namespace.
`setQualifierType`	Invokes the CIM Object Manager to add the specified qualifier type to the specified namespace.

Provider API Package (`com.sun.wbem.provider`)

Provider APIs are interfaces the CIM Object Manager and object providers use to communicate with each other. Providers can use these interfaces to provide the CIM Object Manager dynamic data.

When an application requests dynamic data from the CIM Object Manager, the CIM Object Manager uses these interfaces to pass the request to the provider. Providers are classes that perform the following functions in response to a request from the CIM Object Manager:

- Map information from a managed device to CIM Java classes
  - Get information from a device
  - Pass the information to the CIM Object Manager in the form of CIM Java classes
- Map the information from CIM Java classes to managed device format
  - Get the required information from the CIM Java class
  - Pass the information to the device in native device format

The following table describes the interfaces in the Provider package.

TABLE 5–4 Provider Interfaces

Interface	Description
`CIMProvider`	Base interface implemented by all providers.
`InstanceProvider`	Base interface implemented by instance providers. Instance providers serve dynamic instances of classes.
`MethodProvider`	Interface implemented by method providers, which provide implementation for all methods of CIM classes.
`PropertyProvider`	Interface implemented by property providers, which are used to retrieve and update dynamic properties. Dynamic data is not stored in the CIM Object Manager Repository.

CHAPTER 6

Writing Client Applications

This chapter explains how to use the Client Application Programming Interfaces (APIs) to write client applications.

- Overview
- Opening and Closing a Client Connection
- Working with Instances
- Enumerating Objects
- Calling Methods
- Retrieving Class Definitions
- Handling Exceptions
- Advanced Programming Topics
- Sample Programs

For detailed information on the CIM and Client APIs, see the Javadoc reference pages.

Overview

A Web-Based Enterprise Management (WBEM) application is a standard Java program that uses Sun WBEM SDK APIs to manipulate CIM objects. A client application typically uses the CIM API to construct an object (for example, a namespace, class, or instance) and then initialize that object. The application then uses the Client APIs to pass the object to the CIM Object Manager and request a WBEM operation, such as creating a CIM namespace, class, or instance.

Sequence of a Client Application

Sun WBEM SDK applications typically follow this sequence:

1. Connect to the CIM Object Manager - (`CIMClient`).

   A client application contacts a CIM Object Manager to establish a connection each time it needs to perform a WBEM operation, such as creating a CIM Class or updating a CIM instance.

2. Use one or more APIs to perform some programming tasks.

   Once a program connects to the CIM Object Manager, it uses the APIs to request operations.

3. Close the client connection to the CIM Object Manager - (`close`).

   Applications should close the current client session when finished. Use the `CIMClient` interface to close the current client session and free any resources used by the client session.

Example — Typical Sun WBEM SDK Application

Code Example 6–1 is a simple application that connects to the CIM Object Manager, using all default values. The program gets a class and then enumerates and prints the instances in that class.

CODE EXAMPLE 6–1 Typical Sun WBEM SDK Application

```
import java.rmi.*;
import com.sun.wbem.client.CIMClient;
import com.sun.wbem.cim.CIMInstance;
import com.sun.wbem.cim.CIMValue;
import com.sun.wbem.cim.CIMProperty;
import com.sun.wbem.cim.CIMNameSpace;
import com.sun.wbem.cim.CIMObjectPath;
import com.sun.wbem.cim.CIMClass;
import com.sun.wbem.cim.CIMException;
import java.util.Enumeration;

/**
 * Gets the class specified in the command line (args[1]). Gets the
 * instances of the class in the namespace specified in the command
 * line (args[0]).
 */
1 public class WBEMsample {
2    public static void main(String args[]) throws CIMException {
3        CIMClient cc = null;
4        try {
5           /* args[0] contains the namespace. We create
```

(continued)

(Continuation)

```
6        a namespace object (cns) to store the namespace. */
7        CIMNameSpace cns = new CIMNameSpace(args[0]);
8        /* Connect to the CIM Object manager and pass it
9       the namespace object containing the namespace. */
10      cc = new CIMClient(cns);
11      /* args[1] contains the class name. We create a
12      CIM Object Path that references the specified
13       class in the current namespace. */
14      CIMObjectPath cop = new CIMObjectPath(args[1]);
15      /* Get the class object referenced by the CIM Object
16      Path. */
17      cc.getClass(cop);
18      //Deep enumeration of the class and all its subclasses
19      Enumeration e = cc.enumInstances(cop, true);
20      while(e.hasMoreElements()) {
21        CIMObjectPath op = (CIMObjectPath)e.nextElement();
22        System.out.println(op);
23      }
24        catch (Exception e) {
25               System.out.println("Exception: "+e);
26       }
27        if(cc != null) {
28              cc.close();
29       }
30    }
31 }
```

Typical Programming Tasks

Once a client application connects to the CIM Object Manager, it uses the API to request operations. The program's feature set determines which operations it needs to request. The typical tasks that most programs perform are:

- Working with instances – creating, deleting, and updating
- Enumerating objects
- Calling methods
- Retrieving class definitions
- Handling errors

In addition, applications may occasionally perform the following tasks:

- Creating namespaces
- Deleting namespaces

- Creating a class
- Deleting a class
- Working with qualifiers

Opening and Closing a Client Connection

The first task an application performs is to open a client session to a CIM Object Manager. WBEM Client applications request object management services from a CIM Object Manager. The client and CIM Object Manager can run on the same hosts or on different hosts. Multiple clients can establish connections to the same CIM Object Manager.

This section describes some basic concepts about namespaces and explains how to use:

- The `CIMClient` class to connect to the CIM Object Manager
- The `close` method to close the client connection

Using Namespaces

Before writing an application, you need to understand the CIM concept of a namespace. A namespace is a directory-like structure that can contain other namespaces, classes, instances, and qualifier types. The names of objects within a namespace must be unique. All operations are performed within a namespace. The installation of Solaris WBEM Services creates two namespaces:

- `root\cimv2` – Contains the default CIM classes that represent objects on the system on which Solaris WBEM Services is installed. This is the default namespace.
- `root\security` – Contains the security classes

When an application connects to the CIM Object Manager, it must either connect to the default namespace (`root\cimv2`) or specify another namespace, for example, `root\security` or a namespace you created.

Once connected to the CIM Object Manager in a particular namespace, all subsequent operations occur within that namespace. An application can connect to a namespace within a namespace. This is similar to changing to a subdirectory within

a directory. Once the application connects to the new namespace, all subsequent operations occur within that namespace.

Connecting to the CIM Object Manager

A client application contacts a CIM Object Manager to establish a connection each time it needs to perform a WBEM operation, such as creating a CIM class or updating a CIM instance. The application uses the `CIMClient` class to create an instance of the client on the CIM Object Manager. The `CIMClient` class takes three optional arguments:

- namespace

  The host name and namespace to use for this client connection. The default is `root\cimv2` on the local host.

- user name

  The name of a valid Solaris user account. The CIM Object Manager checks the access privileges for this user to determine what type of access to CIM objects is allowed. The default user account is `guest`.

- password

  The password for this user account. The password must be a valid password for the user's Solaris account. The default password is `guest`.

Once connected to the CIM Object Manager, all subsequent `CIMClient` operations occur within the specified namespace.

Examples — Connecting to the CIM Object Manager

The following examples show two ways of using the `CIMClient` interface to connect to the CIM Object Manager.

In Code Example 6–2, the application takes all the default values. That is, it connects to the CIM Object Manager running on the local host (the same host the client application is running on), in the default namespace (`root\cimv2`), using the default user account and password, `guest`.

CODE EXAMPLE 6–2 Connecting to the Default Namespace

```
/* Connect to root\cimv2 namespace on the local
host as user guest with password guest

cc = new CIMClient();
```

(continued)

In Code Example 6–3, the application connects to namespace `A` on host `happy`. The application first creates an instance of a namespace to contain the string name of the namespace (`A`). Next the application uses the `CIMClient` class to connect to the CIM Object Manager, passing it the namespace object, user name, and host name.

CODE EXAMPLE 6–3 Connecting to a Non-Default Namespace

```
/* Create a namespace object initialized with A
(name of namespace) on host happy.
CIMNameSpace cns = new CIMNameSpace("happy", A);

// Connect to the namespace as user Mary.
cc = new CIMClient(cns, "Mary", "");
```

Closing a Client Connection

Applications should close the current client session when finished. Use the `close` method to close the current client session and free any resources used by the client session.

The following sample code closes the client connection. The instance variable *cc* represents this client connection.

```
cc.close();
```

Working with Instances

This section describes how to create a CIM instance, delete a CIM instance, and update an instance (get and set the property values of one or more instances).

Creating an Instance

Use the `newInstance` method to create an instance of an existing class. If the existing class has a key property, an application must set it to a value that is guaranteed to be unique. As an option, an instance can define additional qualifiers that are not defined for the class. These qualifiers can be defined for the instance or for a particular property of the instance and do not need to appear in the class declaration.

Applications can use the `getQualifiers` method to get the set of qualifiers defined for a class.

Example — Creating an Instance

The code segment in Code Example 6–4 uses the `newInstance` method to create a Java class representing a CIM instance (for example, a Solaris package) from the `Solaris_Package` class.

CODE EXAMPLE 6–4 Creating an Instance (`newInstance()`)

```
/*Connect to the CIM Object Manager in the root\cimv2 namespace
on the local host. */
CIMClient cc = new CIMClient();

// Get the Solaris_Package class
cimclass = cc.getClass(newCIMObjectPath("Solaris_Package");

/* Create a new instance of the Solaris_Package class,
 populated with the default values for properties. If the provider
 for the class does not specify default values, the values of the
 properties will be null and must be explicitly set. */
 ci = cimclass.newInstance();
```

Deleting an Instance

Use the `deleteInstance` method to delete an instance.

Example — Deleting an Instance

The example in Code Example 6–5 connects the client application to the CIM Object Manager and uses the following interfaces to delete all instances of a class:

- `CIMObjectPath` to construct an object containing the CIM object path of the object to be deleted
- `enumInstance` to get the instances and all instances of its subclasses
- `deleteInstance` to delete each instance

CODE EXAMPLE 6–5 Deleting an Instance (`deleteInstance`)

```
import java.rmi.*;
import com.sun.wbem.client.CIMClient;
import com.sun.wbem.cim.CIMInstance;
import com.sun.wbem.cim.CIMValue;
import com.sun.wbem.cim.CIMProperty;
import com.sun.wbem.cim.CIMNameSpace;
import com.sun.wbem.cim.CIMObjectPath;
import com.sun.wbem.cim.CIMClass;
import java.util.Enumeration;

public class DeleteInstances {
  public static void main(String args[]) throws CIMException {
   CIMClient cc = null;
   try {
     /* Construct a namespace object containing the
        command line arguments. */
     CIMNameSpace cns = new CIMNameSpace(args[0]);

     /* Pass the namespace object to the CIM Object Manager.*/
       CIMClient cc = new CIMClient(cns);

     /* Construct an object containing the CIM object
        path of the object we want to delete. */
       CIMObjectPath cop = new CIMObjectPath(args[1]);

     /* Do a deep enumeration (deep is set to
            CIMClient.DEEP) of the instances of the object. A deep
            enumeration of the instances of a class returns the
            class instances and all instances of its subclasses. */
     Enumeration e = cc.enumInstances(cop, CIMClient.DEEP);

     // Print the name of each object and delete the instance.
     while(e.hasMoreElements()) {
      CIMObjectPath op = (CIMObjectPath)e.nextElement();
      System.out.println(op);
      cc.deleteInstance(op);
        }
    }
   catch (Exception e) {
        System.out.println("Exception: "+e);
   }

    // If the client connection is open, close it.
   if(cc != null) {
           cc.close();
```

(continued)

```
    }
  }
}
```

Getting and Setting Instances

An application frequently uses the `getInstance` method to retrieve CIM instances from the CIM Object Manager.

Example — Getting Instances

The code segment in Code Example 6–6 lists all processes on a given system. This example uses the `enumInstances` method to get the names of instances of the `CIM_Process` class. Running this code on a Microsoft Windows 32 system returns Windows 32 processes. Running this same code on a Solaris system returns Solaris processes.

CODE EXAMPLE 6–6 Getting Instances of a Class (`getInstance`)

```
{
//Create namespace cns
CIMnameSpace cns = new CIMNameSpace;

//Connect to the cns namespace on the CIM Object Manager
cc = new CIMClient(cns);

/* Pass the CIM Object Path of the CIM_Process class
to the CIM Object Manager. We want to get instances of this class. */

CIMObjectPath op = new CIMObjectPath("CIM_Process");

/* The CIM Object Manager returns a vector of object paths, the names
of instances of the CIM_Process class. */
Vector v = cc.enumInstances(op, true);

/* Iterate through the vector of instance object paths.
Use the CIM Client getInstance interface to get
the instances referred to by each object name. */

for (int i=0; i < v.size(); i++) {
```

(continued)

(Continuation)

```
  // Get the instance
  CIMInstance ci = cc.getInstance(v.elementAt(i));

  /* Get the process ID string for each instance of
  CIM_Process. */
  CIMProperty cp = ci.getProperty("Handle");
}
```

Example — Getting a Property

Code Example 6–7 prints the value of the lockspeed property for all Solaris processes. This code segment uses the following methods:

- `enumInstances` – to get the names of all instances of Solaris processor
- `getInstance` – to get the instance data for each instance name
- `getProperty` – to get the value of the lockspeed for each instance
- `println` – to print the lockspeed value

CODE EXAMPLE 6–7 Printing Processor Information (`getProperty`)

```
/* Connect to the CIM Object Manager as user mary with password
contrary in the /root namespace on myhost */
{
CIMNameSpace cns = new CIMNamesSpace ("myhost" "/root");
cc = new CIMClient (cns, "/root", "mary", "contrary");

// Get names of all instances of Solaris_Processor
Vector op cc.enumInstances("Solaris_Processor")

// For each Solaris processor, get its instance data
while (vector has more elements) {
        cn.getNextElement();
        cc.getInstance (cn);

// Print the lockspeed of each processor
p = ci.getProperty("lockspeed")
        System.out.println(p.getValue().getValue());
}
```

(continued)

Example — Setting Instances

The code segment in Code Example 6–8 gets a CIM instance, updates one of its property values, and passes the updated instances to the CIM Object Manager.

A CIM property is a value used to describe a characteristic of a CIM class. Properties can be thought of as a pair of functions, one to *set* the property value and one to *get* the property value.

CODE EXAMPLE 6–8 Setting Instances (`setInstance`)

```
{
/* Get instances for each element in a vector, update the
property value of b to 10 in each instance,
and pass the updated instance to the CIM Object Manager. */

For (int i=0; i(v.size(); i++) {
  CIMInstance ci = cc.getInstance(v.elementAt(i));
  ci.setProperty("b",new CIMValue(10));
  cc.setInstance(new CIMObjectPath(),ci);

}
```

Enumerating Objects

Enumerating objects means getting a list of the names of the objects. Once you get a list of object names, you can get the instances of that object, its properties, or other information about the object. The Sun WBEM SDK provides APIs for enumerating namespaces, classes, and instances.

The enumeration APIs take two Boolean arguments, *deep* and *shallow*. The behavior of these parameters depends upon the particular method being used. A *deep* enumeration of instances of a class returns the class instances and all instances of its subclasses. A *deep* enumeration of a class returns all subclasses of the class, but does

not return the class itself. A *shallow* enumeration of the instances of a class returns the instances of that class. A *shallow* enumeration of a class returns the direct subclasses of that class.

The following examples show how to use the enumeration APIs to enumerate a namespace and a class.

Example — Enumerating Namespaces

The sample program in Code Example 6–9 uses the `enumNameSpace` method in the CIM Client class to print the names of the namespace and all the namespaces contained within the namespace.

CODE EXAMPLE 6–9 Enumerating Namespaces (`enumNameSpace`)

```
import java.rmi.*;
import com.sun.wbem.client.CIMClient;
import com.sun.wbem.cim.CIMInstance;
import com.sun.wbem.cim.CIMValue;
import com.sun.wbem.cim.CIMProperty;
import com.sun.wbem.cim.CIMNameSpace;
import com.sun.wbem.cim.CIMObjectPath;
import com.sun.wbem.cim.CIMClass;
import java.util.Enumeration;

/ **
  * This program takes a namespace argument and calls the
  * enumNameSpace CIM Client interface to get a list of the
  * namespaces within the namespace specified by the CIMObjectPath.
  * and all the namespaces contained in the namespace
  * (CIMClient.DEEP).  The program then prints the name of the specified
  * namespace (CIMClient.SHALLOW).
 /**

public class EnumNameSpace {

  // EnumNameSpace takes a string of arguments
  public static void main (String args[ ]) {
   CIMClient cc = null;

   try {
   // Create a namespace object for the namespace passed as an argument
   CIMNameSpace cns = new CIMNameSpace(args[0], "");

   //
 Connect to the CIM Object Manager in the namespace passed as an argument
   CIMClient cc = new CIMClient(cns);

   // Create an object path to store the namespace name on the current host
   CIMObjectPath cop = new CIMObjectPath("",args[1]);
```

(continued)

(Continuation)

```
   // Enumerate the namespace and all namespaces it contains
      // (deep is set to CIMClient.DEEP)
   Enumeration e = cc.enumNameSpace(cop, CIMClient.DEEP);

   // Iterate through the list of namespaces and print each name.
   for (; e.hasMoreElements();
      System.out.println(e.nextElement()));
      System.out.println("++++++");
   // Iterate through the list of namespaces (CIMClient.SHALLOW) and
      // print each name.
   e = cc.enumNamesSpace(cop, CIMClient.SHALLOW);
   for (; e.hasMoreElements();
     System.out.println(e.nextElement()));
 }

 // Catch and print any exception returned
  catch (Exception e) {
    System.out.println("Exception: "+e);
   }

// If the client connection is open, close it.
  if(cc != null) {
            cc.close();
     }

  }
}
```

Example — Enumerating Classes

A Java GUI application might use the code segment in Code Example 6–10 to display a list of classes and subclasses to a user. Once the user selects a particular class, the code enumerates the class.

CODE EXAMPLE 6–10 Enumerating Classes (`enumClass`)

```
/* Creates an object containing the path of a CIM object. */
CIMObjectPath (op = new(CIMObjectPath());

/* Specifies the object path name as A. */
cop.setName("A");

/* Vector returns the object path of the object, classes, and
```

(continued)

(Continuation)

```
all subclasses within those classes. The object path includes the
namespace, class name, and keys (if the object is an instance). */

/* This vector contains the CIM Object Paths to the enumerated
classes. */
Vector v = cc.enumClass(cop, true);
```

Calling Methods

Use the `invokeMethod` method to call a method in a class supported by a provider. To retrieve the signature of a method, an application must first get the definition of the class to which the method belongs. The `invokeMethod` interface takes four arguments:

Data Type	**Description**
CIMObjectPath	The name of the instance on which the method must be invoked.
String	The name of the method to call.
Vector	Input parameters to pass to the method.
Vector	Output parameters to get from the method.

The `invokeMethod` method returns a CIMValue. The return value is null when the method you invoke does not define a return value.

Example — Calling a Method

The code segment in Code Example 6–11 gets the instances of the `CIM_Service` class (services that manage device or software features) and uses the `invokeMethod` method to stop each service.

CODE EXAMPLE 6–11 Calling a Method (`invokeMethod`)

```
{


/* Pass the CIM Object Path of the CIM_Service class
to the CIM Object Manager. We want to get instances of this class. */

CIMObjectPath op = new CIMObjectPath("CIM_Service");

/* The CIM Object Manager returns a vector of object paths, the names
of instances of the CIM_Service class. */

Vector v = cc.enumInstances(op, true);

/* Iterate through the vector of instance object paths.
Use the CIM Client getInstance interface to get
the instances referred to by each object name. */

for (int i=0; i < v.size(); i++) {

   // Get the instance

   CIMInstance ci = cc.getInstance(v.elementAt(i));

   //Invoke the Stop Service method to stop the CIM services.

  c.invokeMethod(v.element(i), "StopService", null, null);

}

}
```

Retrieving Class Definitions

Use the `getClass` method to get a CIM class.

Example — Retrieving a Class Definition

The code shown in Code Example 6–12, uses the following methods to retrieve a class definition:

- `CIMNameSpace` – to create a new namespace
- `CIMClient` – to create a new client connection to the CIM Object Manager
- `CIMObjectPath` – to create an object path, an object to contain the name of the class to retrieve
- `getClass` – to retrieve the class from the CIM Object Manager

CODE EXAMPLE 6–12 Retrieving a Class Definition (`getClass`)

```
import java.rmi.*;
import com.sun.wbem.client.CIMClient;
import com.sun.wbem.cim.CIMInstance;
import com.sun.wbem.cim.CIMValue;
import com.sun.wbem.cim.CIMProperty;
import com.sun.wbem.cim.CIMNameSpace;
import com.sun.wbem.cim.CIMObjectPath;
import com.sun.wbem.cim.CIMClass;
import com.sun.wbem.cim.CIMException;
import java.util.Enumeration;

/**
 * Gets the class specified in the command line. Works in the default
 * namespace /root/cimv2.
 */
public class GetClass {
    public static void main(String args[]) throws CIMException {
        CIMClient cc = null;
        try {
            CIMNameSpace cns = new CIMNameSpace(args[0]);

            cc = new CIMClient(cns);

            CIMObjectPath cop = new CIMObjectPath(args[1]);
            cc.getClass(cop);
        }
        catch (Exception e) {
            System.out.println("Exception: "+e);
        }
        if(cc != null) {
            cc.close();
        }
    }
}
```

Handling Exceptions

Each interface has a throws clause that defines a CIM Exception. An exception is an error condition. The CIM Object Manager uses Java exception handling and creates a hierarchy of WBEM-specific exceptions. The `CIMException` class is the base class for CIM exceptions. All other CIM exception classes extend from the `CIMException` class.

Each class of CIM exceptions defines a particular type of error condition that API code handles. See Table 5–2 for a description of the CIM exception APIs.

Using the Try / Catch Clauses

The Client API uses standard Java try / catch clauses to handle exceptions. Generally, an application catches exceptions and either takes some corrective action or passes some information about the error to the user.

The CIM rules are not explicitly identified in the CIM specification. In many cases, they are implied by example. In many cases, the error code refers to a general problem, for example, a data type mismatch, but the programmer must figure out what the correct data type is for the data.

Syntactic and Semantic Error Checking

The MOF Compiler (`mofc`) compiles `.mof` text files into Java classes (bytecode). The MOF Compiler does syntactical checking of the `MOF` files. The CIM Object Manager does semantic and syntactical checking because it can be accessed by many different applications.

The MOF file in Code Example 6–13 defines two classes, A and B. If you compiled this example file, the CIM Object Manager would return a semantic error because only a key can override another key.

CODE EXAMPLE 6–13 Semantic Error Checking

```
Class A         \\Define Class A

{     [Key]
int a;
}

Class B:A       \\Class B extends A
```

(continued)

(Continuation)

```
{  [overrides ("c", key (false) ]
int b;
}
```

Advanced Programming Topics

This section describes advanced programming operations and operations that you would use less frequently.

Creating a Namespace

The installation compiles the standard CIM MOF files into the default namespaces, `/root/cimv2` and `/root/security`. If you create a new namespace, you must compile the appropriate CIM MOF files into the new namespace before creating objects in it. For example, if you plan to create classes that use the standard CIM elements, compile the CIM Core Schema into the namespace. If you plan to create classes that extend the CIM Application Schema, compile the CIM Application into the namespace.

Example — Creating a Namespace

The code segment in Code Example 6–14 uses a two-step process to create a namespace within an existing namespace.

- First, it uses the `CIMNameSpace` method to construct a namespace object. This namespace object contains the parameters to be passed to the CIM Object Manager when the namespace is actually created.
- Second, the example uses the `CIMClient` class to connect to the CIM Object Manager and pass it the namespace object. The CIM Object Manager creates the namespace, using the parameters contained in the namespace object.

CODE EXAMPLE 6–14 Creating a Namespace (`CIMNameSpace`)

```
{
/*Creates a namespace object on the client, which stores the
parameters passed to it. args[0] contains the host name (for example,
myhost); args[1] contains the namespace (for example,
the toplevel directory.) */
CIMNameSpace cns = new CIMNameSpace (args[0], args[1]);
/* Connects to the CIM Object Manager and passes it the
namespace object (cns) containing the namespace parameters. */
CIMClient cc = new CIMClient (cns);
/* Passes to the CIM Object Manager another namespace object that
contains a null string (host name) and args[2], the name of a name
space (for example, secondlevel). */
CIMNameSpace cop = new CIMNameSpace("", args[2]);
/* Creates a new namespace called secondlevel under the
toplevel namespace on myhost./*
cc.createNameSpace(cop);
}
```

Deleting a Namespace

Use the `deleteNameSpace` method to delete a namespace.

Example — Deleting a Namespace

The code segment in Code Example 6–15 first creates a namespace and then uses the `deleteNameSpace` method to delete it.

CODE EXAMPLE 6–15 Deleting a Namespace (`deleteNameSpace`)

```
{
/* Creates a namespace object on the client to contain the
namespace parameters, args[0] (host name) and args[1]
(namespace name). */

CIMNameSpace cns = new CIMNameSpace (args[0], args[1]);

/* Connects to the CIM Object Manager and passes it
the namespace object. */

CIMClient cc = new CIMClient (cns);

/* Passes the CIM Object Manager a namespace object containing a null
```

(continued)

(Continuation)

```
host argument (we are not changing the CIM Object Manager host) and
the name of the namespace to be deleted. */

CIMNameSpace cop = new CIMNameSpace("", args[2]);

/* Delete namespace cop. */

cc.deleteNameSpace(cop);
```

Creating a Base Class

Applications can create classes using either the MOF language or the client APIs. If you are familiar with MOF syntax, use a text editor to create a MOF file and then use the MOF Compiler to compile it into Java classes. This section describes how to use the client APIs to create a base class.

Use the `CIMClass` class to create a Java class representing a CIM class. To declare the most basic class, you need only specify the class name. Most classes include properties that describe the data of the class. To declare a property, include the property's data type, name, and an optional default value. The property data type must be an instance of `CIMDataType` (one of the predefined CIM data types).

A property can have a *key* qualifier, which identifies it as a key property. A key property uniquely defines the instances of the class. Only keyed classes can have instances. Therefore, if you do not define a key property in a class, the class can only be used as an abstract class.

If you define a key property in a class in a new namespace, you must first compile the core MOF files into the namespace. The core MOF files contain the declarations of the standard CIM qualifiers, such as the *key* qualifier. For more information on MOF files, see Chapter 3.

Class definitions can be more complicated, including such MOF features as aliases, qualifiers, and qualifier flavors.

Example — Creating a CIM Class

The example in Code Example 6–16 creates a new CIM class in the default namespace (`/root/cimv2`) on the local host. This class has two properties, one of which is the key property for the class. The example then uses the `newInstance` method to create an instance of the new class.

CODE EXAMPLE 6–16 Creating a CIM Class (`CIMClass`)

```
{
/* Connect to the /root/cimv2 namespace
on the local host and create a new class called myclass */
// Connect to the default namespace on local host.
CIMClient cc = new CIMClient();
// Construct a new CIMClass object
CIMClass cimclass = new CIMClass();
// Set CIM class name to myclass.
cimclass.setName("myclass");
// Construct a new CIM property object
CIMProperty cp = new CIMProperty();
// Set property name
cp.setName("keyprop");
// Set property type
cp.setType(CIMDatatype.getpredefined(CIMDataType.STRING);
// Construct a new CIM Qualifier object
CIMQualifier cq = new CIMQualifier();
// Set the qualifier name
cq.setName("key");
// Add the new key qualifier to the property
cp.addQualfiier(cq);

/* Create an integer property initialized to 10 */
// Construct a new CIM property object
CIMProperty mp = new CIMProperty();
// Set property name to myprop
mp.setName("myprop");
// Set property type
mp.setType(CIMDatatype.getpredefined(CIMDataType.INTEGER);
// Initialize myprop to 10
mp.setValue(CIMValue.setValue(10));

/* Add the new properties to myclass and call the CIM Object Manager
to create the class. */
// Add the key property to class object
cimclass.addProperty(cp);
// Add the integer property to class object
cimclass.addProperty(mp);

/* Connect to the CIM Object Manager and pass the new class */
cc.setClass(new CIMObjectPath(),cimclass);
// Pass the new class to the CIM Object Manager
ci = cc.newInstance();
// Create a new CIM instance of myclass
// If the client connection is open, close it.
  if(cc != null) {
            cc.close();
}
```

Deleting a Class

Use the `CIMClient deleteClass` method to delete a class. Deleting a class removes the class, its subclasses, and all instances of the class; it does not delete any associations that refer to the deleted class.

Example — Deleting a Class

The example in Code Example 6–17 uses the `deleteClass` interface to delete a class.

CODE EXAMPLE 6–17 Deleting a Class (`deleteClass`)

```
import java.rmi.*;
import com.sun.wbem.client.CIMClient;
import com.sun.wbem.cim.CIMInstance;
import com.sun.wbem.cim.CIMValue;
import com.sun.wbem.cim.CIMProperty;
import com.sun.wbem.cim.CIMNameSpace;
import com.sun.wbem.cim.CIMObjectPath;
import com.sun.wbem.cim.CIMClass;
import com.sun.wbem.cim.CIMException;
import java.util.Enumeration;

/**
 * Deletes the class specified in the command line. Works in the default
 * namespace root\cimv2.
 */
public class DeleteClass {
    public static void main(String args[]) throws CIMException {
        CIMClient cc = null;
        try {
            CIMNameSpace cns = new CIMNameSpace(args[0]);

            cc = new CIMClient(cns);

            CIMObjectPath cop = new CIMObjectPath(args[1]);
            cc.deleteClass(cop);
        }
        catch (Exception e) {
            System.out.println("Exception: "+e);
        }
        if(cc != null) {
            cc.close();
        }
    }
}
```

Working with Qualifier Types and Qualifiers

A CIM qualifier is an element that characterizes a CIM class, instance, property, method, or parameter. Qualifiers have the following attributes:

- Type
- Value
- Name

In Managed Object Format syntax, each CIM qualifier must have a CIM qualifier type declared in the same MOF file. Qualifiers do not have a scope attribute. Scope indicates which CIM elements can use the qualifier. Scope can only be defined in the qualifier type declaration; it cannot be changed in a qualifier.

The following sample code shows the MOF syntax for a CIM qualifier type declaration. This statement defines a qualifier type named `key`, with a Boolean data type (default value false), which can describe only a property and a reference to an object. The `DisableOverride` flavor means that key qualifiers cannot change their value.

```
Qualifier Key : boolean = false, Scope(property, reference),
        Flavor(DisableOverride);
```

The following sample code shows the MOF syntax for a CIM qualifier. In this sample MOF file, `key` and description are qualifiers for the property `test`. The property data type is an integer with the value `a`.

```
{
[key, Description("test")]
int a
}
```

Example — Getting CIM Qualifiers

The code segment in Code Example 6–18 uses the `CIMQualifier` class to identify the CIM qualifiers in a vector of CIM elements. The example returns the property name, value, and type for each CIM Qualifier.

A qualifier flavor is a flag that governs the use of a qualifier. Flavors describe rules that specify whether a qualifier can be propagated to derived classes and instances and whether or not a derived class or instance can override the qualifier's original value.

CODE EXAMPLE 6–18 Getting CIM Qualifiers (`CIMQualifier`)

```
...

    } else if (tableType == QUALIFIER_TABLE) {
  CIMQualifier prop = (CIMQualifier)cimElements.elementAt(row);
  if (prop != null) {
      if (col == nameColumn) {
   return prop.getName();
      } else if (col == typeColumn) {
   CIMValue cv = prop.getValue();
   if (cv != null) {
       return cv.getType().toString();
   } else {
       return "NULL";         }
    }

...
```

Example — Setting CIM Qualifiers

Code Example 6–19 is a code segment that sets a list of CIM qualifiers for a new class to the qualifiers in its superclass.

CODE EXAMPLE 6–19 Set Qualifiers (`setQualifiers`)

```
   ...
 try  {
      cimSuperClass = cimClient.getClass(new CIMObjectPath(scName));
      Vector v = new Vector();
      for (Enumeration e = cimSuperClass.getQualifiers().elements();
     e.hasMoreElements();) {
     CIMQualifier qual = (CIMQualifier)((CIMQualifier)e.nextElement()).clone();
     v.addElement(qual);
      }
      cimClass.setQualifiers(v);
  } catch (CIMException exc) {
      return;
  }
}
```

Sample Programs

The examples directory contains sample programs that use the client API to perform a function. You can use these examples to start writing your own applications more quickly. The sample programs are described in Chapter 8.

To run a sample program, type the command:

`java` *program_name*

For example, `java createNameSpace`.

CHAPTER 7

Writing a Provider Program

This chapter describes how to write a provider, including the following topics:

- About Providers
- Implementing a Provider Interface
- Installing a Provider
- Registering a Provider
- Modifying a Provider
- Provider Examples

For detailed information on the provider APIs, see the Javadoc reference pages.

About Providers

Providers are classes that communicate with managed objects to access data. Providers forward this information to the CIM Object Manager for integration and interpretation. When the CIM Object Manager receives a request from a management application for data that is not available from the CIM Object Manager Repository, it forwards the request to a provider.

Object providers must be installed on the same machine as the CIM Object Manager. The CIM Object Manager uses object provider application programming interfaces (APIs) to communicate with locally installed providers.

When an application requests dynamic data from the CIM Object Manager, the CIM Object Manager uses the provider interfaces to pass the request to the provider.

Providers perform the following functions in response to a request from the CIM Object Manager:

- Map the native information format to CIM Java classes
  - Get information from a device
  - Pass the information to the CIM Object Manager in the form of CIM Java classes
- Map the information from CIM Java classes to native device format
  - Get the required information from the CIM Java class
  - Pass the information to the device in native device format

Types of Providers

Providers are categorized according to the types of requests they service. The Sun WBEM SDK supports three types of providers:

- Instance – Supply dynamic instances of a given class, for example, Solaris packages. Instance providers support one or more of the following operations:
  - Instance retrieval
  - Enumeration
  - Modification
  - Deletion
- Property – Supply dynamic property values, for example, disk space.
- Method – Supply methods of one or more classes. A method is a function that describes the behavior of a class. Methods must be implemented by a provider.

A single provider can support both methods and instances, which can be convenient.

Most providers are pull providers, which means they maintain their own data, generating it dynamically when necessary. Pull providers have minimal interaction with the CIM Object Manager and the CIM Repository. The data managed by a pull provider typically changes frequently, requiring the provider to either generate the data dynamically or retrieve it from a local cache whenever an application issues a request. A provider can also contact the CIM Object Manager.

A single provider can act simultaneously as a class, instance, and method provider by proper registration and implementation of all relevant methods.

Implementing a Provider Interface

Providers implement a provider interface that supports the type of service specific to their role. In order to implement the interface, a provider class must first declare the interface in an `implements` clause, and then it must provide an implementation (a body) for all of the abstract methods of the interface. The following table describes the provider interfaces. You can include a method provider, instance provider, and property provider in a single Java class file, or store each provider in a separate file.

Providers can communicate with the CIM Object Manager using the `initialize` method. The `initialize` method takes an argument of type `CIMOMhandle`, which is a reference to the CIM Object Manager. The `CIMOMhandle` class contains methods that providers can use to transfer data to and from the CIM Object Manager.

TABLE 7–1 Provider Interfaces

Interface	Description
`CIMProvider`	Base interface implemented by all providers.
`InstanceProvider`	Base interface implemented by instance providers. Instance providers serve dynamic instances of classes.
`MethodProvider`	Interface implemented by method providers, which provide implementation for all methods of CIM classes.
`PropertyProvider`	Interface implemented by property providers, which are used to retrieve and update dynamic properties. Dynamic data is not stored in the CIM Object Manager Repository.

The Instance Provider Interface (`InstanceProvider`)

The following table describes the methods in the instance provider interface in the Provider package (`com.sun.wbem.provider`).

These methods each take the *op* argument, the CIM object path of the specified CIM class or CIM instance. The object path includes the namespace, class name, and keys (if the object is an instance). The namespace is a directory that can contain other namespaces, classes, instances, and qualifier types. A key is a property that uniquely identifies an instance of a class. Key properties have a *KEY* qualifier.

For example, the following object path has two parts:

```
\\myserver\root\cimv2\Solaris_ComputerSystem:Name=mycomputer:
   CreationClassName=Solaris_ComputerSystem
```

- `\\myserver\root\cimv2`

  The default CIM namespace on host `myserver`.

- `Solaris_ComputerSystem:Name=mycomputer:`
  `CreationClassName=Solaris_ComputerSystem`

  A specific Solaris Computer System object in the default namespace on host `myserver`. This Solaris computer system is uniquely identified by two key property values in the format (property=value):

  - `Name=mycomputer`
  - `CreationClassName=Solaris_ComputerSystem`

TABLE 7–2 `InstanceProvider` Interface Methods

Method	**Description**
`enumInstances`	Enumerates all instances of the class specified in the object path. You can do deep or shallow enumeration, but currently the CIM Object Manager only requests shallow enumeration.
`getInstance`	Returns the instance specified in the object path (*op*).
`setInstance`	Sets the instance specified in the object path (*op*). If the instance does not exist, it must be added.
`deleteInstance`	Deletes the instance specified in the object path (*op*).

Example — Implementing an Instance Provider

The code segment in Code Example 7–1, creates a Solaris instance provider class that routes requests for instance data from the CIM Object Manager to one or more specialized providers. These specialized providers service requests for dynamic data for a particular type of Solaris object. For example, the `Solaris_Package` provider services requests for instances of the `Solaris_Package` class.

An instance provider must implement all methods in the `InstanceProvider` interface. The code segment in Code Example 7–1 shows only two methods:

- `enumInstances` – Calls the appropriate provider to enumerate Solaris packages and patches. This method does a deep enumeration, which returns the class instances and all instances of its subclasses.
- `getInstances` – Calls the appropriate provider to get instances of Solaris packages and patches.

CODE EXAMPLE 7–1 Implementing an Instance Provider

```
public class Solaris implements InstanceProvider
{
   /**
    * Top-level provider class routes requests from the CIM
    * Object Manager to the appropriate provider.
    */

    public void initialize(CIMONHandle, ch)
    throws CIMException {
    }

    public void cleanup()
    throws CIMException {
    }

    /* This class returns a vector of enumerated instances of the
    specified object in the specified class. If the object is a
    Solaris package, it calls the Solaris_Package provider to return
    a list of the Solaris packages on the system. If the object is a
    Solaris patch, it calls the Solaris_Patch provider to return a
    list of the Solaris patches on the system. */

    public Vector enumInstances(CIMObjectPath op, CIMClient.DEEP, CIMClass cc)
    throws CIMException {
        if (op.getObjectName().equalsIgnoreCase("solaris_package")) {
            Solaris_Package sp = new Solaris_Package();
            return sp.enumerateInstances(op);
        }
        if (op.getObjectName().equalsIgnoreCase("solaris_patch")) {
            Solaris_Patch sp = new Solaris_Patch();
            return sp.enumerateInstances(op);
        }
       return new Vector();
```

(continued)

(Continuation)

```
    }


    /* This class returns an instance of the specified object in the
    specified class. If the object is a Solaris package, it
    calls the Solaris_Package provider to return the data for the
    specified Solaris package. If the object is a Solaris patch, it
    calls the Solaris_Patch provider to return the data for the
    specified Solaris patch. */

    public CIMInstance getInstance(CIMObjectPath op, CIMClass cc)
    throws CIMException {
        if (op.getObjectName().equalsIgnoreCase("solaris_package")) {
            Solaris_Package sp = new Solaris_Package();
            return sp.getInstance(op,cc);
        }
        if (op.getObjectName().equalsIgnoreCase("solaris_patch")) {
            Solaris_Patch sp = new Solaris_Patch();
            return sp.getInstance(op,cc);
        }
}
```

The specialized Solaris instance providers use the API to get and set instances of objects. These providers also declare native methods that call C functions to get Solaris-specific values, such as host name, serial number, release, machine, architecture, and manufacturer.

The code segment in Code Example 7–2 shows the `solaris_package` class, which is called in Code Example 7–1. This code segment implements the `getInstance` method. This method creates a new instance of the specified class and then fills it with properties returned from native C functions, such as `GetPkgArchitecture( )`.

CODE EXAMPLE 7–2 Solaris Package Provider

```
public class Solaris_Package
{
    public CIMInstance getInstance(CIMObjectPath op, CIMClass cc) {
        String pkgName = "";

        for (Enumeration e = op.getKeys().elements(); e.hasMoreElements();) {
            CIMProperty cp = (CIMProperty)e.nextElement();
            if (cp.getName().equalsIgnoreCase("name")) {
                pkgName = (String) ((CIMValue)(cp.getValue())).getValue();
```

(continued)

(Continuation)

```
            }
        }


        CIMInstance ci = cc.newInstance();
        ci.setProperty("Name", new CIMValue(pkgName));
        ci.setProperty("TargetOperatingSystem",
                        new CIMValue(new UnsignedInt16(29)));

        ci.setProperty("Status",
                     new CIMValue(GetPkgStatus(pkgName)));
      ci.setProperty("Architecture",
                        new CIMValue(GetPkgArchitecture(pkgName)));
        ci.setProperty("Description", new CIMValue(GetPkgDescription(pkgName)));
        ci.setProperty("Caption", new CIMValue(GetPkgDescription(pkgName)));
        ci.setProperty("Manufacturer", new CIMValue(GetPkgVendor(pkgName)));
        ci.setProperty("Category", new CIMValue(GetPkgCategory(pkgName)));
        ci.setProperty("Basedir", new CIMValue(GetPkgBasedir(pkgName)));
       return ci;
    }

    native String GetPkgDescription(String pkgName);
    native String GetPkgArchitecture(String pkgName);
    native String GetPkgVersion(String pkgName);
    native String GetPkgVendor(String pkgName);
    native String GetPkgBasedir(String pkgName);
    native String GetPkgCategory(String pkgName);
    native String GetPkgStatus(String pkgName);


   static {
       System.loadLibrary("NativeUnix");

    }

}
```

The Property Provider Interface (`PropertyProvider`)

The following table describes the methods in the property provider interface.

TABLE 7–3 `PropertyProvider` Interface Methods

Method	Description
`getPropertyValue`	Returns the value of the property for the specified instance.
`setPropertyValue`	Sets the value of the property for the specified instance

Example — Implementing a Property Provider

The code segment in Code Example 7–3 creates a property provider (`fruit_prop_provider`) class that is registered in Code Example 7–5. The `fruit_prop_provider` implements the `PropertyProvider` interface.

This sample property provider illustrates the `getPropertyValue` method, which returns a property value for the specified class, parent class, and property name. A CIM property is defined by its name and origin class. Two or more properties can have the same name, but the origin class uniquely identifies the property.

CODE EXAMPLE 7–3 Implementing a Property Provider

```
fruit_prop_provider implements PropertyProvider
{
public CIMValue getPropertyValue(CIMObjectpath op, string originclass,
 string PropertyName){

  if (PropertyName.euqals("a")
   return new CIMValue("fooa")

  else
   return new CIMValue("foob");
}
...
}
```

The Method Provider Interface (`MethodProvider`)

The `MethodProvider` method takes the following arguments:

- *op* – Object path to the instance whose method must be invoked.
- *originClass* – Name of the class in which this method was originally defined in the class hierarchy. CIM properties are attributes of a CIM class that are inherited from a parent class. A CIM property is uniquely identified within a namespace by its name and origin class. For example, two properties named `speed` can be distinguished by their respective origin classes `DiskDrive` and `CPU`.
- *inParams* – Vector of `CIMValues` that are the input parameters for the method.
- *CIMValue* – Return value of the method. If the method has no return value, it must return null. A CIM value stores the CIM data type and value of a CIM property. CIM data types (defined in the CIM Specification) are limited to intrinsic data types. The following table provides the WBEM data type name for each CIM data type.

TABLE 7–4 Sun WBEM SDK and CIM Data Type Names

CIM Data Type	WBEM Data Type	Description
`uint8`	`UnsignedInt8`	Unsigned 8-bit integer
`sint8`	`Byte`	Signed 8-bit integer
`uint16`	`UnsignedInt16`	Unsigned 16-bit integer
`sint16`	`Short`	Signed 16-bit integer
`uint32`	`UnsignedInt32`	Unsigned 32-bit integer
`sint32`	`Integer`	Signed 32-bit integer
`uint64`	`UnsignedInt64`	Unsigned 64-bit integer
`sint64`	`Long`	Signed 64-bit integer
`string`	`String`	UCS-2 string
`boolean`	`Boolean`	Boolean
`real32`	`Float`	IEEE 4-byte floating point
`real64`	`Double`	IEEE 8-byte floating point
`datetime`	`CIMDateTime`	String containing a date-time

TABLE 7–4 Sun WBEM SDK and CIM Data Type Names *(continued)*

CIM Data Type	WBEM Data Type	Description
`classname` ref	`CIMObjectPath`	Strongly typed reference
`char16`	`Character`	16-bit UCS-2 character

The following table describes the method in the Method Provider interface.

TABLE 7–5 `MethodProvider` Interface Methods

Method	Description
`invokeMethod`	The CIM Object Manager calls this method when the specified method is invoked.

Example — Implementing a Method Provider

The code segment in Code Example 7–4 creates a Solaris provider class that routes requests to execute methods from the CIM Object Manager to one or more specialized providers. These specialized providers service requests for dynamic data for a particular type of Solaris object. For example, the `Solaris_Package` provider services requests to execute methods in the `Solaris_Package` class.

The method provider in this example implements a single method `invokeMethod` that calls the appropriate provider to perform one of following operations:

- Reboot a Solaris system
- Reboot or shut down a Solaris system
- Delete a Solaris serial port

CODE EXAMPLE 7–4 Implementing a Method Provider

```
public class Solaris implements MethodProvider
{

    public void initialize(CIMONHandle, ch)
```

(continued)

(Continuation)

```
    throws CIMException {
    }

    public void cleanup()
    throws CIMException {
    }
   public CIMValue invokeMethod(CIMObjectPath op, String methodName,
        Vector inParams, Vector outParams) throws CIMException {
        if (op.getObjectName().equalsIgnoreCase("solaris_computersystem")) {
            Solaris_ComputerSystem sp = new Solaris_ComputerSystem();
            if (methodName.equalsIgnoreCase("reboot")) {
               return new CIMValue (sp.Reboot());
            }
        }
        if (op.getObjectName().equalsIgnoreCase("solaris_operatingsystem")) {
            Solaris_OperatingSystem sos = new Solaris_OperatingSystem();
            if (methodName.equalsIgnoreCase("reboot")) {
               return new CIMValue (sos.Reboot());
            }
            if (methodName.equalsIgnoreCase("shutdown")) {
               return new CIMValue (sos.Shutdown());
            }
        }
        if (op.getObjectName().equalsIgnoreCase("solaris_serialport")) {
            Solaris_SerialPort ser = new Solaris_SerialPort();
            if (methodName.equalsIgnoreCase("disableportservice")) {
               return new CIMValue (ser.DeletePort(op));
            }
        }
        return null;
    }
}
```

Writing a Native Provider

Providers get and set information on managed devices. A native provider is a machine-specific program written to run on a managed device. For example, a provider that accesses data on a Solaris system will most likely include C functions to query the Solaris system. Two common reasons for writing a native provider are:

- Efficiency – You may want to implement a small portion of time-critical code in a lower-level programming language, such as assembly, and then have your Java application call these functions.
- Need to access platform-specific features – The standard Java class library may not support the platform-dependent features needed by your application.

- Legacy code - Often, you have legacy code written in some programming language other than Java and want to continue to use the code with a Java provider.

The Java Native Interface is the native programming interface for Java that is part of the JDK. By writing programs using the JNI, you ensure that your code is completely portable across all platforms. The JNI allows Java code that runs within a Java Virtual Machine (VM) to operate with applications and libraries written in other languages, such as C, C++, and assembly.

For more information on writing and integrating Java programs with native methods, visit the Java web site at `http://www.javasoft.com/docs/books/tutorial/native1.1/index.html`.

Installing a Provider

After you write a Provider, you must specify the location of the provider class files and any shared library files.

▼ How to Install a Provider Program

1. **Specify the location of shared library files in one of the following ways:**
   - Set the LD_LIBRARY_PATH environment variable to the location of the shared library files. For example, using the C shell:

     ```
     % setenv LD_LIBRARY_PATH /wbem/provider/
     ```

     For example, using the Borne shell:

     ```
     % set LD_LIBRARY_PATH = /wbem/provider/
     ```

   - Copy the shared library files to the directory specified by the LD_LIBRARY_PATH environment variable. The installation sets this environment variable to */install_dir*`/opt/SUNWconn/wbem/lib`. For example:

```
% cp libnative.so /install_dir/opt/SUNWconn/wbem/lib
% cp native.c /install_dir/opt/SUNWconn/wbem/lib
```

2. **Move the provider class files to** */install_dir*`/opt/SUNWconn/wbem/bin`**.**

3. **Set the CLASSPATH variable to the directory that contains the provider class files.**

   For example, if you put the provider files in:

   /install_dir`/opt/SUNWconn/wbem/bin/com/mycomp/wbem/provider/`

   you would set your CLASSPATH variable as follows, using the C shell:

```
% setenv CLASSPATH  com/mycomp/wbem/provider/myprovider
```

   For the Borne shell, use the following syntax:

```
$ set CLASSPATH = com/mycomp/wbem/provider/myprovider
```

4. **Make sure the CIM Object Manager is running.**

   The installation starts the CIM Object Manager. If it is not running, see “Restarting the CIM Object Manager” on page 150.

Registering a Provider

Providers register with the CIM Object Manager to publish information about the data and operations they support and their physical implementation. The CIM Object Manager uses this information to load and initialize the provider and to determine the right provider for a particular client request. All types of providers follow the same procedure for registration. Neither the CIM Object Manager or the provider needs to be running during the registration process.

▼ How To Register a Provider

1. **Create a MOF file defining a CIM class.**

2. **Assign the *provider* qualifier to the class. Assign a provider name to the *provider* qualifier.**

   The provider name identifies the Java class to serve as the provider for this class. You must specify the complete class name. For example:

   Note - We recommend following Java class and package naming conventions for providers so that provider names are unique. The prefix of a unique package name is always written in all-lowercase ASCII letters and should be one of the top-level domain names, currently com, edu, gov, mil, net, org, or one of the English two-letter codes identifying countries as specified in ISO Standards 3166, 1981.

   Subsequent components of the package name vary according to an organizations own internal naming conventions. Such conventions might specify that certain directory name components be division, department, project, machine, or login names, for example `com.mycompany.wbem.myprovider`.

```
[Provider("com.kailee.wbem.providers.provider_name")]
Class_name {
...
};
```

3. **Compile the MOF file. For example:**

   `mofcomp` *class_name*

Example — Registering a Provider

The sample MOF file in Code Example 7–5 creates a class called `Fruit` that registers an instance provider (`fruit_class_provider`), a property provider (`fruit_prop_provider`), and a method provider (`fruit_method_provider`).

CODE EXAMPLE 7–5 Registering an Instance, Property, and Method Provider

```
// Registers fruit_class_provider as the provider for the Fruit class
[Provider("com.food.fruitprovider.fruit_class_provider")]

Fruit {
```

(continued)

(Continuation)

```
// fruit_prop_provider is the provider for property a
  [Provider("com.food.fruitprovider.fruit_class_provider")]
-  string a;

  // fruit_prop_provider is also the provider for property b
  [Provider("com.food.fruitprovider.fruit_class_provider")]
  string b;

  // fruit_method_provider is the provider for method b
  [Provider("com.food.fruitprovider.fruit_class_provider")]
  int b();
};
```

Modifying a Provider

You can make changes to a Provider class while the CIM Object Manager and provider are running. But you must stop and restart the CIM Object Manager to make the changes take effect.

▼ To Modify a Provider

1. **Edit the provider source file.**

2. **Compile the provider source file. For example:**

   `% java` *MyProvider.java*

3. **Become root on your system by typing the following command at the system prompt:**

```
% su
```

4. **Type the root password when you are prompted.**

5. **Change directories to the location of the `init.wbem` command by typing the following command:**

```
# cd /etc/init.d/
```

6. **Stop the CIM Object Manager by typing the following command:**

```
# init.wbem -stop
```

7. **Restart the CIM Object Manager by typing the following command:**

```
# init.wbem -start
```

Provider Examples

Chapter 8 explains how to set up and run the Provider examples.

CHAPTER 8

Using Sun WBEM SDK Examples

This chapter describes the sample programs provided in the Sun WBEM SDK and includes the following topics:

- About Example Programs
- Using the Client Examples
- Using the Provider Examples

About Example Programs

The Sun WBEM SDK provides example Java programs, which are installed in */install_dir*/SUNWconn/wbem/demo. You can use the source code as a basis for developing your own programs. Two types of example programs are provided:

- Client programs – Programs that use the client and CIM application programming interfaces (APIs) to request WBEM operations from the CIM Object Manager
- Provider programs – Programs that communicate with managed objects to access data

Using Client Examples

The client examples use the client APIs to create, delete, and list classes, instances, and namespaces. The five types of client programs are:

- Enumeration – Enumerates classes and instances. It does deep and shallow enumeration on a class that is passed from the command line.
- Logging – Writes and reads log records
- Miscellaneous – Deletes classes and instances.
- Namespace – Creates and deletes namespaces.
- System Information – Displays Solaris process information for the system and network you select.

Client Example Files

The following table describes the example client program files and lists the commands and arguments to run each example.

TABLE 8–1 Client Example Files

Example File Name	Description	Command to Run
`CreateNameSpace`	Connects to the CIM Object Manager as the specified user and creates a namespace on the specified host. You must specify the user name and password of the administrative account for the CIM Object Manager Repository.	`java CreateNameSpace` *host parentNS childNS username password*
`DeleteNameSpace`	Deletes the specified namespace on the specified host. You must specify the user name and password of the administrative account for the CIM Object Manager Repository.	`java DeleteNameSpace` *host parentNS childNS username password*

TABLE 8–1 Client Example Files *(continued)*

Example File Name	Description	Command to Run
`ClientEnum`	Enumerates classes and instances in the specified class in the default namespace `/root/cimv2` on the specified host.	`java ClientEnum` *host className*
`CreateLog`	Creates a log record on the specified host. You must specify the user name and password of the administrative account for the CIM Object Manager Repository..	`java CreateLog` *host username password*
`ReadLog`	Reads a log record on the specified host. You must specify the user name and password of the administrative account for the CIM Object Manager Repository..	`java ReadLog` *host username password*
`DeleteClass`	Deletes the specified class in the default namespace `/root/cimv2` on the specified host. You must specify the user name and password of the administrative account for the CIM Object Manager Repository.	`java DeleteClass` *host className username password*

TABLE 8–1 Client Example Files *(continued)*

Example File Name	Description	Command to Run
`DeleteInstances`	Deletes instances of the specified class in the default namespace `/root/cimv2` on the specified host. You must specify the user name and password of the administrative account for the CIM Object Manager Repository.	`java DeleteInstances` *host className username password*
`SystemInfo`	Displays Solaris processor and system information for the specified host in a window.	`java SystemInfo` *host*

Running the Client Examples

To run a client example program, type the command:

`java` *program_name*

Most of the example programs take required arguments that have default values. For example, the `CreateNameSpace` example program takes five arguments:

- Host name
- Parent namespace
- Child namespace
- User name
- Password

Use the following syntax to specify default values for command line arguments.

Argument	Default Value	Syntax
Host name	`local host`	`.`
Parent namespace	`/root/cimv2`	`" "`

Argument	Default Value	Syntax
Child namespace	`Null`	`" "`
User name	`GUEST`	`" "`
Password	`GUEST`	`" "`

The following example runs the `CreateNameSpace` example, which connects to the default `root\cimv2` namespace on the local host as user `admin` with password `secret`.

```
java CreateNameSpace . "" admin secret
```

Using the Provider Examples

The example provider is a Java program that returns system properties and prints the string, “Hello World.” The provider calls native C methods to execute the code and return the values to the provider.

Provider Example Files

The following table describes the files that make up the example Provider program.

TABLE 8–2 Provider Example Files

File	Purpose
`NativeProvider`	Top level provider program that fulfills requests from the CIM Object Manager and routes them to the `Native_Example` provider. The `NativeProvider` program implements the `instanceProvider` and `methodProvider` APIs, and declares methods that enumerate instances and get an instance of the `Native_Example` class. It also declares a method that invokes a method to print the string "Hello World."
`Native_Example.mof`	Creates a class that registers the `NativeProvider` provider with the CIM Object Manager. The `Native_Example.mof` file identifies `NativeProvider` as the provider to service requests for dynamic data in the `Native_Example` class. This MOF file also declares the properties and methods to be implemented by the `NativeProvider.`
`Native_Example.java`	The `NativeProvider` program calls this provider to implement methods that enumerate instances and get an instance of the `Native_Example` class. The `Native_Example` provider uses the APIs to enumerate objects and create instances of objects. The `Native_Example` class declares native methods, which call C functions in the `native.c` file to get system-specific values, such as host name, serial number, release, machine, architecture, and manufacturer.
`native.c`	C program that implements calls from the `Native_Example` Java provider in native C code.
`Native_Example.h`	Machine-generated header file for `Native_Example` class. Defines the correspondence between the Java native method names and the native C functions that execute those methods.
`libnative.so`	Binary native C code compiled from the `native.c` file.

Writing a Native Provider

For detailed information on writing and integrating Java programs with native methods, visit the Java Web page at `http://www.javasoft.com/docs/books/tutorial/native1.1/TOC.html`.

Setting Up the Provider Example

The example provider program, `NativeProvider`, enumerates instances and gets properties for instances of the `Native_Example` class. You can use the CIM WorkShop to view this class and its instances.

▼ How to Set Up the Provider Example

1. **Specify the location of shared library files in one of the following ways:**
   - Set the LD_LIBRARY_PATH environment variable to the location of the shared library files. For example:

     ```
     % set LD_LIBRARY_PATH /install_dir/SUNWconn/wbem/demo/provider/jni/
     ```

   - Copy the shared library files to the directory specified by the LD_LIBRARY_PATH environment variable. Installation sets this environment variable to */install_dir*`/opt/SUNWconn/wbem/lib`. For example:

```
% cp libnative.so /install_dir/opt/SUNWconn/wbem/lib
% cp native.c /install_dir/opt/SUNWconn/wbem/lib
% cp Native_Example.h /install_dir/opt/SUNWconn/wbem/lib
```

2. **Move the provider class files to the directory containing the CIM Object Manager. For example:**

   ```
   % mv Native*.class /install_dir/opt/SUNWconn/wbem/bin
   ```

3. **Make sure the CIM Object Manager is running.**

The installation starts the CIM Object Manager. If it is not running, see "Restarting the CIM Object Manager" on page 150.

4. **Compile the** `Native_Example.mof` **file. For example:**

```
% mofcomp Native_Example.mof
```

Compiling this MOF file loads the `Native_Example` class in the CIM Object Manager and identifies `NativeProvider` as its provider.

5. **Run CIM WorkShop and view the** `Native_Example` **class. For example:**

```
% /opt/SUNWconn/wbem/bin/cimworkshop &
```

6. **In the Toolbar, click the Find Class icon.**

7. **In the Input dialog box, type** `Native_Example` **and click OK.**

CHAPTER 9

Error Messages

This chapter discusses the error messages generated by components of Solaris WBEM Services and the Sun WBEM SDK. The following topics are covered.

- How Error Messages are Generated
- Parts of Error Messages
- Finding Information About Error Messages
- Generated Error Messages

How Error Messages are Generated

The CIM Object Manager generates initial error messages that are used by both the MOF Compiler and the CIM WorkShop. The MOF Compiler appends a line to the error message indicating where in a `.mof` file the error occurred.

Parts of Error Messages

Error messages are made up of the following parts:

- Unique identifier – Character string that differentiates the error message from other error messages
- Exception message – Explanation of the error message
- Parameters – Placeholders for the specific classes, methods, and qualifiers that are cited in the exception message

Error Message Example

For example, the MOF Compiler may return the following error message:

```
REF_REQUIRED = Association class CIM_Docked needs
 at least two refs. Error in line 12.
```

- `REF_REQUIRED` is the unique identifier.
- `Association class CIM_Docked needs at least two refs` is the exception message.
- `CIM_Docked` is a parameter. A parameter can be replaced with the name of any appropriate class, property, method, or qualifier.

For Developers: Error Message Templates

WBEM provides exception templates for all possible error messages in the `ErrorMessages_en.properties` file of the APIs. In an exception template that requires parameters, the first parameter is represented as `{0}` and the second parameter is represented as `{1}`.

The following exception template is used in the previous example:

```
REF_REQUIRED = Association class {0} needs at least two refs.
```

Finding Information About Error Messages

You can search for the unique identifier of an error message in the *Javadoc* reference pages to receive an explanation about the content of the error message.

The following section provides a detailed explanation of each error message. The error messages are organized by unique identifiers. For each error message, the following types of information are provided, when applicable:

- Unique identifier, displayed as a heading.

- Description of the parameters used in the error message.
- Example of the error message as it is displayed to a user. This example is provided if the error message uses parameters to show how the error message will be displayed when elements, such as a class name, are substituted for the parameters.
- Cause, or reason why the error message was generated, and background or referential information that is helpful for understanding the error message.
- Solution, including steps you can take to resolve the error are provided when available.

Generated Error Messages

The following section lists and describes the error messages generated by the MOF Compiler, CIM Object Manager, and CIM WorkShop.

`ABSTRACT_INSTANCE`

Description

The `ABSTRACT_INSTANCE` error message uses one parameter, `{0}`, which is replaced by the name of the abstract class.

Example

`ABSTRACT_INSTANCE` = Abstract class `ExampleClass` cannot have instances.

Cause

Instances were programmed for the specified class. However, the specified class is an abstract class, and abstract classes cannot have instances.

Solution

Remove the programmed instances.

`CHECKSUM_ERROR`

Description

The `CHECKSUM_ERROR` error message uses no parameters.

Example

`CHECKSUM_ERROR` = Checksum not valid.

Cause

The message could not be sent because it was damaged or corrupted. The damage could have occurred accidentally in transit or by a malicious third party.

Note - This error message is displayed when the CIM Object Manager receives an invalid checksum. A checksum is the number of bits in a packet of data passed over the network. This number is used by the sender and the receiver of the information to ensure that the transmission is secure and that the data has not been corrupted or intentionally modified during transit.

An algorithm is run on the data before transmission, and the checksum is generated and included with the data to indicate the size of the data packet. When the message is received, the receiver can recompute the checksum and compare it to the sender's checksum. If the checksums match, the transmission was secure and the data was not corrupted or modified.

Solution

Resend the message using Solaris WBEM Services security features. For information about Solaris WBEM Services security, see Chapter 12.

`CIM_ERR_ACCESS_DENIED`

Description

The `CIM_ERR_ACCESS_DENIED` error message does not use parameters.

Example

`CIM_ERR_ACCESS_DENIED` = Insufficient privileges.

Cause

This error message is displayed when a user does not have the appropriate privileges and permissions to complete an action.

Solution

See your CIM Object Manager administrator to request privileges to complete the operation.

`CIM_ERR_ALREADY_EXISTS`

Instance 1: CIM_ERR_ALREADY_EXISTS

Description

This instance of the `CIM_ERR_ALREADY_EXISTS` error message uses one parameter, `{0}`, which is replaced by the name of the duplicate class.

Example

`CIM_ERR_ALREADY_EXISTS` = Duplicate class `CIMRack`

Cause

The class you attempted to create uses the same name as an existing class.

Solution

In CIM WorkShop, search for existing classes to see the class names that are in use, then create the class using a unique class name.

Instance 2: CIM_ERR_ALREADY_EXISTS

Description

This instance of the `CIM_ERR_ALREADY_EXISTS` error message uses one parameter, `{0}`, which is replaced by the name of the duplicate instance.

Example

`CIM_ERR_ALREADY_EXISTS` = Duplicate instance `SolarisRack`

Cause

The instance for a class you attempted to create uses the same name as an existing instance.

Solution

In CIM WorkShop, search for existing instances to see the names that are in use, then create the instance using a unique name.

Instance 3: CIM_ERR_ALREADY_EXISTS

Description

This instance of the `CIM_ERR_ALREADY_EXISTS` error message uses one parameter, `{0}`, which is replaced by the name of the duplicate namespace.

Example

`CIM_ERR_ALREADY_EXISTS` = Duplicate namespace /root/CIMV2

Cause

The namespace you attempted to create uses the same name as an existing namespace.

Solution

In CIM WorkShop, search for existing namespaces to see the names that are in use, then create the namespace using a unique name.

Instance 4: CIM_ERR_ALREADY_EXISTS

Description

This instance of the `CIM_ERR_ALREADY_EXISTS` error message uses one parameter, `{0}`, which is replaced by the name of the duplicate qualifier type.

Example

`CIM_ERR_ALREADY_EXISTS` = Duplicate qualifier type `Key`

Cause

The qualifier type you attempted to create uses the same name as an existing qualifier type of the property it modifies.

Solution

In CIM WorkShop, search for qualifier types that exist for the property to see the names that are in use, then create the qualifier type using a unique name.

`CIM_ERR_FAILED`

Description

The `CIM_ERR_FAILED` error message uses one parameter, `{0}`, which is replaced by a character string, a message that explains the error condition and its possible cause.

Example

`CIM_ERR_FAILED`=Invalid entry.

Cause

The `CIM_ERR_FAILED` error message is a generic message that can be displayed for a large number of different error conditions.

Solution

Because `CIM_ERR_FAILED` is a generic error message, many types of conditions can cause the message. The solution varies depending on the error condition.

`CIM_ERR_INVALID_PARAMETER`

Description

The `CIM_ERR_INVALID_PARAMETER` error message uses one parameter, `{0}`, which is replaced by the name of the class that is missing a schema prefix.

Example

`CIM_ERR_INVALID_PARAMETER` = Class `System` has no schema prefix.

Cause

A class was created without providing a schema prefix in front of the class name. The Common Information Model requires that all classes are provided with a schema prefix. For example, classes developed as part of the CIM Schema require a `CIM` prefix: `CIM_Container`. Classes developed as part of the Solaris Schema require a `Solaris` prefix: `Solaris_System`.

Solution

Provide the appropriate schema prefix for the class definition. Find all instances of the class missing the prefix and replace them with the class name and prefix.

`CIM_ERR_INVALID_SUPERCLASS`

Description

The parameter CIM_ERR_INVALID_SUPERCLASS uses two parameters:

- {0} is replaced by the name of the specified subclass.
- {1} is replaced by the name of the class for which a specified subclass does not exist.

Example

CIM_ERR_INVALID_SUPERCLASS = Superclass CIM_Chassis for class CIM_Container does not exist.

Cause

A class is specified to belong to a particular superclass, but the superclass does not exist. The specified superclass may be misspelled, or a non-existent superclass name may have been specified accidentally in place of the intended superclass name. Or, the superclass and the subclass may have been interpolated: the specified superclass actually may be a subclass of the specified subclass. In the previous example, CIM_Chassis is specified as the superclass of CIM_Container, but CIM_Chassis is a subclass of CIM_Container.

Solution

Check the spelling and the name of the superclass to ensure it is correct. Ensure that the superclass exists in the namespace.

CIM_ERR_NOT_FOUND

Instance 1: CIM_ERR_NOT_FOUND

Description

This instance of the CIM_ERR_NOT_FOUND error message uses one parameter, {0}, which is replaced by the name of the non-existent class.

Example

CIM_ERR_NOT_FOUND = Class Solaris_Device does not exist.

Cause

A class is specified, but it does not exist. The specified class may be misspelled, or a non-existent class name may have been specified accidentally in place of the intended class name.

Solution

Check the spelling and the name of the class to ensure that it is correct. Ensure that the class exists in the namespace.

Instance 2: CIM_ERR_NOT_FOUND

Description

This instance of the error message `CIM_ERR_NOT_FOUND` uses two parameters:

- `{0}` is replaced by the name of the specified instance
- `{1}` is replaced by the name of the specified class

Example

`CIM_ERR_NOT_FOUND` = Instance `Solaris_EnterpriseData` does not exist for class `Solaris_ComputerSystem`.

Cause

The instance does not exist.

Solution

Create the instance.

Instance 3: CIM_ERR_NOT_FOUND

Description

This instance of the `CIM_ERR_NOT_FOUND` error message uses one parameter, `{0}`, the name of the specified namespace.

Example

`CIM_ERR_NOT_FOUND` = Namespace `verdant` does not exist.

Cause

The specified namespace is not found. This error may occur if the name of the namespace was entered incorrectly due to a typing error or spelling mistake.

Solution

Retype the name of the namespace. Ensure that typing and spelling are correct.

`CLASS_REFERENCE`

Description

The `CLASS_REFERENCE` error message uses two parameters.

- `{0}` parameter is replaced by the name of the class that was defined to participate in a reference.
- `{1}` parameter is replaced by the name of the reference.

Example

`CLASS_REFERENCE` = Class `SolarisExample1` must be declared as an association to have reference `SolarisExample2`

Cause

A property was defined for a class to indicate that the class has a reference. However, the class is not part of an association relationship. A class can only be defined to have a reference as a property if it participates in an association relationship with another class.

Solution

Create the association relationship, then set up the reference to the association as a property of this class.

`INVALID_CREDENTIAL`

Description

The `INVALID_CREDENTIAL` error message does not use parameters.

Example

`INVALID_CREDENTIAL` = Invalid credentials.

Cause

This error message is displayed when an invalid password has been entered.

Solution

If you receive this message from CIM WorkShop, delete the invalid password from the Password field of the CIM WorkShop authentication dialog box and type the password again. If this error message was received from the MOF Compiler, at the system prompt, log in again and type the correct password. Ensure that you spell the password correctly.

`INVALID_QUALIFIER_NAME`

Description

The `INVALID_QUALIFIER_NAME` error message uses one parameter, `{0}`, which is replaced by the Managed Object Format notation that depicts an empty qualifier name.

Example

`INVALID_QUALIFIER_NAME` = Invalid qualifier name ‘‘ ’’

Cause

A qualifier was created for a property, but a qualifier name was not specified.

Solution

Include the qualifier name in the context of the qualifier definition.

`KEY_OVERRIDE`

Description

The KEY_OVERRIDE error message uses two parameters:

- {0} parameter is replaced by the name of the non-abstract class that is put in an override relationship with a class that has one or more Key qualifiers.
- {1} parameter is replaced by the name of the concrete class that has the Key qualifier.

Example

KEY_OVERRIDE = Non-key Qualifier SolarisCard cannot override key Qualifier SolarisLock.

Cause

A non-abstract class, referred to as a concrete class, is put into an override relationship with a concrete class that has one or more Key qualifiers. In CIM, all concrete classes require at least one Key qualifier, and a non-Key class cannot override a class that has a Key.

Solution

Create a Key qualifier for the non-Key class.

KEY_REQUIRED

Description

The KEY_REQUIRED error message uses one parameter, {0} which is replaced by the name of the class that requires a key.

Example

KEY_REQUIRED = Concrete (non-abstract) class ClassName needs at least one key.

Cause

A Key qualifier was not provided for a concrete class. In CIM, all non-abstract classes, referred to as concrete classes, require at least one Key qualifier.

Solution

Create a Key qualifier for the class.

METHOD_OVERRIDDEN

Description

The METHOD_OVERRIDDEN command uses three parameters:

- {0} replaced by the name of the method that is trying to override the method represented by parameter {1}.
- {1} is replaced by the name of the method that has already been overridden by the method represented by parameter {2}.
- {2} is replaced by the name of the method that has overridden parameter {1}.

Example

METHOD_OVERRIDDEN = Method Resume () cannot override Stop() which is already overridden by Start()

Cause

A method is specified to override another method that has already been overridden by a third method. Once a method has been overridden, it cannot be overridden again.

Solution

Specify a different method to override.

NEW_KEY

Description

The NEW KEY error message uses two parameters.

- {0} is replaced by the name of the key.
- {1} is replaced by the name of the class that is trying to define a new key.

Example

NEW_KEY = Class CIM_PhysicalPackage cannot define new key [Key]

Cause

A class is trying to define a new key when keys already have been defined in a superclass. Once keys have been defined in a superclass, new keys cannot be introduced into the subclasses.

Solution

No action can be taken.

NO_CIMOM

Description

The NO_CIMOM error message uses one parameter, {0} which is replaced by the name of the host that is expected to be running the CIM Object Manager.

Example

NO_CIMOM = CIMOM molly not detected.

Cause

The CIM Object Manager is not running on the specified host.

Solution

Ensure that the CIM Object Manager is running on the host to which you are trying to connect. If the CM Object Manager is not running on that host, connect to a host running the CIM Object Manager.

NO_INSTANCE_PROVIDER

Description

The NO_INSTANCE_PROVIDER error message uses two parameters:

- {0} is replaced by the name of the class for which the instance provider cannot be found.
- {1} is replaced by the name of the instance provider class that was specified.

Example

NO_INSTANCE_PROVIDER = Instance provider RPC_prop for class RPC_Agent not found.

Cause

The Java class of the specified instance provider is not found. This error message indicates that the class path of the CIM Object Manager does not contain one or more of the following:

- Name of the provider class
- Parameters of the provider class
- CIM class for which the provider is defined

Solution

Set the CIM Object Manager environment variable.

NO_METHOD_PROVIDER

Description

The NO_METHOD_PROVIDER error message uses two parameters:

- {0} is replaced by the name of the class for which the method provider cannot be found.
- {1} is replaced by the name of the method provider class that was specified.

Example

NO_METHOD_PROVIDER = Method provider Start_prop for class RPC_Agent not found.

Cause

The Java class of the specified method provider is not found. This error message indicates that the class path of the CIM Object Manager does not contain one or more of the following:

- Name of the provider class
- Parameters of the provider class
- CIM class for which the provider is defined

Solution

Set the CIM Object Manager class path.

`NO_OVERRIDDEN_METHOD`

Description

The error message `NO_OVERRIDDEN_METHOD` uses two parameters:

- `{0}` is replaced by the name of the method that has overridden the method represented by `{1}`.
- `{1}` is replaced by the name of the method that has been overridden.

Example

`NO_OVERRIDDEN_METHOD` = Method `Write` overridden by `Read` does not exist in class hierarchy.

Cause

The method of a subclass is trying to override the method of the superclass, but the method of the superclass already has been overridden by a method that belongs to another subclass. The overridden method that you are trying to override does not exist in the class hierarchy because it has never been defined.

When you override a method, you override its implementation and its signature.

Solution

Ensure that the method exists in the superclass.

`NO_OVERRIDDEN_PROPERTY`

Description

The `NO_OVERRIDDEN_PROPERTY` error message uses two parameters.

- `{0}` is replaced by the name of the property that has overridden `{1}`.
- `{1}` is replaced by the name of the overriding property.

Example

NO_OVERRIDDEN_PROPERTY = Property A overridden by B does not exist in class hierarchy.

Cause

The property of a subclass is trying to override the property of the superclass, but it doesn't succeed because the property of the superclass already has been overridden. The property that you are trying to override does not exist in the class hierarchy.

Solution

Ensure that the property exists in the superclass.

NO_PROPERTY_PROVIDER

Description

The NO_PROPERTY_PROVIDER error message uses two parameters:

- {0} is replaced by the name of the class for which the property provider cannot be found.
- {1} is replaced by the name of the property provider class that was specified.

Example

NO_PROPERTY_PROVIDER = Property provider Write_prop for class RPC_Agent not found.

Cause

The Java class of the specified property provider is not found. This error message indicates that the class path of the CIM Object Manager does not contain one or more of the following:

- Name of the provider class
- Parameters of the provider class
- CIM class for which the provider is defined

Solution

Set the CIM Object Manager class path.

NO_QUALIFIER_VALUE

Description

The NO_QUALIFIER_VALUE error message uses two parameters:

- {0} is replaced by the name of the qualifier that modifies the element {1}

- {1} is the element to which the qualifier refers. Depending on the qualifier, {1} can be a class, property, method, or reference.

Example

NO_QUALIFIER_VALUE = Qualifier [SOURCE] for Solaris_ComputerSystem has no value.

Cause

A qualifier was specified for a property or method, but values were not included for the qualifier. For example, the qualifier VALUES requires a string array to be specified. If the VALUES qualifier is specified without the required string array, the NO_QUALIFIER_VALUE error message is displayed.

Solution

Specify the required parameters for the qualifier. For information about what attributes are required for which qualifiers, see the CIM Specification by the Distributed Management Task Force at the following URL: http://dmtf.org/spec/cims.html.

NO_SUCH_METHOD

Description

The NO_SUCH_METHOD error message uses two parameters:

- {0} is replaced by the name of the specified method
- {1} is replaced by the name of the specified class

Example

NO_SUCH_METHOD = Method Configure() does not exist in class Solaris_ComputerSystem

Cause

Most likely, the method was not defined for the specified class. If the method is defined for the specified class, another method name may have been mispelled or typed differently in the definition.

Solution

Define the method as an operation for the specified class. Otherwise, ensure that the method name and class name were typed correctly.

NO_SUCH_PRINCIPAL

Description

The NO_SUCH_PRINCIPAL error message uses one parameter, {0}, which is replaced by the name of the principal, a user account.

Example

NO_SUCH_PRINCIPAL = Principal molly not found.

Cause

The specified user account cannot be found. The user name may have been mistyped upon login, or a user account has not been set up for the user.

Solution

Ensure that the user name is spelled and typed correctly upon login. Ensure that a user account has been set up for the user.

NO_SUCH_QUALIFIER1

Description

The NO_SUCH_QUALIFIER1 error message uses one parameter, {0}, which is replaced by the name of the undefined qualifier.

Example

NO_SUCH_QUALIFIER1 = Qualifier [LOCAL] not found.

Cause

A new qualifier was specified, but was not defined as part of the extension schema. The qualifier is required to be defined as part of the CIM Schema or an extension schema to be recognized as a valid qualifier for a property or method of a particular class.

Solution

Define the qualifier as part of the extension schema or use a standard CIM qualifier. For information about standard CIM qualifiers and the usage of qualifiers in the CIM schema, see the CIM Specification by the Distributed Management Task Force at the following URL: http://www.dmtf.org/spec/cims.html.

NO_SUCH_QUALIFIER2

Description

The NO_SUCH_QUALIFIER2 error message uses two parameters:

- {0} is replaced by the name of the class, property, or method that the qualifier modifies.
- {1} is replaced by the name of the qualifier that cannot be found.

Example

NO_SUCH_QUALIFIER2 = Qualifier [LOCAL] not found for CIM_LogicalElement

Cause

A new qualifier was specified to modify a property or method of a particular class. The qualifier was not defined as part of the extension schema. The qualifier is required to be defined as part of the CIM schema or an extension schema to be recognized as a valid qualifier for a property or method of a particular class.

Solution

Define the qualifier as part of the extension schema or use a standard CIM qualifier. For information about standard CIM qualifiers and the usage of qualifiers in the CIM schema, see the CIM Specification by the Distributed Management Task Force at the URL, http://www.dmtf.org/spec/cims.html .

NO_SUCH_SESSION

Description

The error message NO_SUCH_SESSION uses one parameter, {0}, which is replaced by the session identifier.

Example

NO_SUCH_SESSION = No such session 4002.

Cause

This message is displayed when a session has been infringed upon by an intruder. The CIM Object Manager removes the session when it detects that someone is trying to maliciously change data. For information about Solaris WBEM Services security features, see Chapter 12.

Solution

Ensure that your CIM environment is secure.

NOT_HELLO

Description

The NOT_HELLO error message uses no parameters.

Example

NOT_HELLO = Not a Hello message.

Cause

This error message is displayed if the data in the hello message—the first message sent to the CIM Object Manager—is corrupted, indicating a security breach.

Solution

No action is available in response to this error message. For information about Solaris WBEM Services security features, see Chapter 12.

`NOT_INSTANCE_PROVIDER`

Description

The `NOT_INSTANCE_PROVIDER` error message uses two parameters:

- `{0}` is replaced by the name of the instance for which the `InstanceProvider` interface is being defined.
- `{1}` is replaced by the name of the Java provider class that does not implement the `InstanceProvider` interface. The `InstanceProvider` interface must be implemented to enumerate all instances of the specified class.

Example

`NOT_INSTANCE_PROVIDER` = `device_prop_provider` for class `Solaris_Provider` does not implement `InstanceProvider`.

Cause

The path to the Java provider class specified by the CLASSPATH environment variable does not implement the `InstanceProvider` interface.

Solution

Ensure that the Java provider class present in the class path implements the InstanceProvider interface. Use the following command when you declare the provider: `public Solaris implements InstanceProvider`. For information about how to implement Solaris WBEM Services providers, see Chapter 7.

`NOT_METHOD_PROVIDER`

Description

The `NOT_METHOD_PROVIDER` error message uses two parameters:

- `{0}` is replaced by the name of the method for which the `MethodProvider` interface is being defined. The `MethodProvider` causes a specified method to be implemented in a program and enacted.
- `{1}` is replaced by the name of the Java provider class that does not implement the `MethodProvider` interface.

Example

`NOT_METHOD_PROVIDER` = Provider `device_method_provider` for class `Solaris_Provider` does not implement `MethodProvider`.

Cause

The Java provider class present in the class path does not implement the `MethodProvider` interface.

Solution

Ensure that the Java provider class present in the class path implements the `MethodProvider` interface. Use the following command when you declare the provider: `public Solaris implements MethodProvider`. For information about how to implement Solaris WBEM Services providers, see Chapter 7.

`NOT_PROPERTY_PROVIDER`

Description

The `NOT_PROPERTY_PROVIDER` error message uses two parameters:

- `{0}` is replaced by the name of the method for which the `PropertyProvider` interface is being defined. The `PropertyProvider` interface is required to retrieve the values of the specified property.
- `{1}`is replaced by the name of the Java provider class that does not implement the `PropertyProvider` interface.

Example

`NOT_PROPERTY_PROVIDER` = Provider `device_property_provider` for class `Solaris_Provider` does not implement `PropertyProvider`.

Cause

The Java provider class present in the class path does not implement the PropertyProvider interface.

Solution

Ensure that the Java provider class present in the class path implements the PropertyProvider interface. Use the following command when you declare the provider: `public Solaris implements PropertyProvider`. For information about how to implement Solaris WBEM Services providers, see Chapter 7.

`NOT_RESPONSE`

Description

The `NOT_RESPONSE` error message uses no parameters.

Example

`NOT_RESPONSE` = Not a response message.

Cause

This error message is displayed when the data in a first response message from the CIM Object Manager is corrupted, indicating a security breach.

Solution

No action is available in response to this error message. For information about Solaris WBEM Services security features, see Chapter 12.

`PROPERTY_OVERRIDDEN`

Description

The `PROPERTY_OVERRIDDEN` error message uses three parameters:

- `{0}` is replaced by the name of the property that is trying to override the property represented by parameter `{1}`.
- `{1}` is replaced by the name of the property that already has been overridden.
- `{2}` is replaced by the name of the property that has overridden the property represented by parameter `{1}`.

Example

`PROPERTY_OVERRIDDEN` = Property `Volume` cannot override `MaxCapacity` which is already overridden by `RawCapacity`

Cause

A property is specified to override another method that has already been overridden by a third method. Once a property has been overridden, it cannot be overridden again.

Solution

Specify a different property to override.

`PS_CONFIG`

Description

The `PS_CONFIG` error message uses one parameter, `{0}`, which is replaced by a description of the details that cause the error to occur. The description varies depending on the type of database used for the repository and the type of situation that causes the error message.

Example

`PS_CONFIG` = The persistent store configuration is incorrect or has not been completed. You may need to run the `wbemconfig` script.

Cause

Solaris WBEM Services requires the `wbemconfig` script to be run after installation. The `wbemconfig` script configures the persistent store and compiles the MOF files that provide the CIM and Solaris Schema classes. If the `wbemconfig` script was not run after the Solaris WBEM Services installation, this error message occurs. If the repository was configured after installation and this error message occurs, the database configuration may have become corrupted.

Solution

Run the `wbemconfig` script. For information about the `wbemconfig` script, see “Configuring After Installing in Solaris 7” on page 142 in the chapter Chapter 10.

`PS_UNAVAILABLE`

Description

The `PS_UNAVAILABLE` error message uses one parameter `{0}`, which is replaced by a message that describes why the persistent store became unavailable.

Example

`PS_UNAVAILABLE` = The persistent store is unavailable. The exception thrown by the repository is ’segmentation fault.’

Cause

This error message is displayed when the CIM Repository is unavailable. This situation could occur if the host on which the CIM Repository resides is brought down temporarily for maintenance, or if the host on which the CIM Repository resides becomes damaged and the repository is taken down and then restored on another host.

Solution

If you receive this message while working in the CIM WorkShop, click the icon that causes the CIM WorkShop authentication dialog box to display. Then, in the Host field, type the name of another host that is running the CIM Repository and the CIM Object Manager. Type the namespace in the Namespace field, your user name and password, and log in. If you receive this message when running the MOF Compiler, type the following command to point to another host running the CIM Repository and CIM Object Manager: `mofcomp -c` *hostname* where `mofcomp` is the command to start the MOF Compiler, `-c` is the parameter that enables you to specify a host computer running the CIM Object Manager, and *hostname* is the name of the specified computer.

`QUALIFIER_UNOVERRIDABLE`

Description

The `QUALIFIER_UNOVERRIDABLE` error message uses two parameters:

- `{0}` parameter is replaced by the name of the qualifier that is set with the `DisableOverride` flavor.
- `{1}` parameter is replaced by the name of the qualifier that is set to be disabled by `{0}`.

Example

QUALIFIER_UNOVERRIDABLE = Test cannot override qualifier Standard because it has DisableOverride flavor.

Cause

The ability of the specified qualifier to override another qualifier is disabled because the flavor of the specified qualifier has been set to DisableOverride or Override=False.

Solution

Reset the ability of the qualifier to EnableOverride or to Override=True.

REF_REQUIRED

Description

The REF_REQUIRED error message uses one parameter, {0}, which is replaced by the name of the class specified to participate in an association relationship.

Example

REF_REQUIRED = Association class CIM_Chassis needs at least two refs.

Cause

A class was set up to participate in an association, but no references were cited. The rules of the Common Information Model specify that an association must contain two or more references.

Solution

Set up the references to the class, then set up the association.

SCOPE_ERROR

Description

The SCOPE_ERROR command uses three parameters:

- {0} is replaced by the name of the class the specified qualifier modifies.
- {1} is replaced by the name of the specified qualifier.
- {2} is replaced by the type of attribute that the qualifier modifies.

Example

SCOPE_ERROR = Qualifier [UNITS] for CIM_Container does not have a Property scope.

Cause

A qualifier was specified in a manner that conflicts with the requirements of the CIM Specification. For example, the [READ] qualifier is defined in the CIM Specification to modify a Property. The scope of the [READ] qualifier is the

definition that directs the [READ] qualifier to modify a Property. If the [READ] qualifier is used in a manner other than the direction of its scope—for example, if the [READ] qualifier is specified to modify a Method—the SCOPE_ERROR message is returned.

Note - The CIM Specification defines the types of CIM elements that a CIM qualifier can modify. This definition of the way in which a qualifier can be used is referred to as its scope. Most qualifiers, by definition, have a scope that directs them to modify properties or methods or both. Many qualifiers have a scope that directs them to modify parameters, classes, associations, indications, or schemas.

Solution

Confirm the scope of the specified qualifier. Refer to the section, "1.Qualifiers" of the CIM Specification by the Distributed Management Task Force at the following URL:http://www.dmtf.org/spec/cim_spec_v20 for the standard definitions of CIM qualifiers. Use a different qualifier for the results you want to achieve, or change your program to use the qualifier according to its CIM definition.

SIGNATURE_ERROR

Description

The SIGNATURE_ERROR error message uses no parameters.

Example

SIGNATURE_ERROR = Signature not verified

Cause

This message is displayed when a message is corrupted either accidentally or maliciously. It differs from the checksum error in that the message has a valid checksum, but the signature cannot be verified by the public key of the client. This protection ensures that even though the session key has been compromised, only the initial client which created the session is authenticated.

Solution

No action is provided for this message, which is displayed when a session has been infringed upon by an intruder. For information about Solaris WBEM Services security features, see Chapter 12.

TYPE_ERROR

Description

The TYPE_ERROR error message uses five parameters:

- {0} is replaced by the name of the specified element, such as a property, method, or qualifier.
- {1} is replaced by the name of the class to which the specified element belongs.

- {2} is replaced by the type defined for the element.
- {3} is replaced by the type of value assigned.
- {4} is replaced by the actual value assigned.

Example

TYPE_ERROR = Cannot convert sint16 4 to a string for VolumeLabel in class Solaris_DiskPartition

Cause

The value of a property or method parameter and its defined type are mismatched.

Solution

Match the value of the property or method with its defined type.

UNKNOWNHOST

Description

The UNKNOWNHOST error message uses one parameter, {0}, which is replaced by the name of the host.

Example

UNKNOWNHOST = Unknown host molly

Cause

A call was made to a specified host. The specified host is unavailable or cannot be located. It is possible that the host name was spelled incorrectly. It is also possible that the host computer was moved to a different domain or that the host name has not been registered in the list of hosts that belong to the domain. The host may be temporarily unavailable due to system conditions.

Solution

Check the spelling of the host name. Ensure that no typing errors were made. Use the ping command to ensure that the host computer is responding. Check the system conditions of the host. Ensure that the host belongs to the specified domain.

VER_ERROR

Description

The VER_ERROR error message uses one parameter, {0}, which is replaced by the version number of the running CIM Object Manager.

Example

VER_ERROR = Unsupported version 0.

Cause

The upgraded version of Solaris WBEM Services does not support the current CIM Object Manager.

Solution

Install the supported version.

PART III Solaris WBEM Services

The Solaris WBEM Services software is Sun's implementation of WBEM on the Solaris operating environment. This software provides the following services:

- Management services transfer CIM data between applications, the CIM Repository, and providers.
- Security services allow you to control user access to CIM objects.
- Logging services allow developers to write log records to and read log records from a log file.

CHAPTER 10

Installing Solaris WBEM Services

This section describes Solaris WBEM Services and explains how to install and remove it from your system. Topics covered include the following:

- About Solaris WBEM Services
- Installation Prerequisites
- Shared Packages
- Installing Solaris WBEM Services
- Configuring After Installing on Solaris 7
- Removing Solaris WBEM Services

About Solaris WBEM Services

Solaris WBEM Services includes the following components:

- CIM Object Manager
- Sun WBEM User Manager
- Solaris Provider

CIM Object Manager

The Common Information Model (CIM) Object Manager manages CIM objects and routes object data. CIM Object Manager is a standard executable Java class file that is started automatically as part of the post-installation process. CIM objects are

represented internally as Java classes. When an application uses the client API to request or update information about a managed object, the CIM Object Manager contacts either the appropriate provider for that object or the CIM Repository, the persistent storage mechanism.

Classes, properties, and methods handled by a provider have a Provider qualifier that identifies the provider to contact for the class. When the CIM Object Manager receives a request for a class that has a Provider qualifier, it routes the request to the specified provider. If no provider is specified, it routes the request to the persistent data storage, using the Java Naming and Directory Interface (JNDI).

The CIM Object Manager can be installed and run on one or more Solaris hosts. When a WBEM-enabled client connects to a CIM Object Manager, it gets a reference to the CIM Object Manager. The client can then perform WBEM operations using this reference.

Semantic and Syntactic Checking

CIM Object Manager performs syntactical and semantic checking. Syntactical checking refers to the ability to detect an error, such as a misplaced semicolon or a forgotten brace, in a line of code. Semantic checking refers to the ability to detect an error in the rules or logic of the program. The CIM Object Manager follows rules provided by the Common Information Model, and detects deviations from CIM rules in a WBEM application.

For example, CIM rules designate that only a key property can override another key property. Class A, which is assigned a key, cannot be overwritten by Class B because Class B is not assigned a key. In this case, CIM Object Manager returns a semantic error.

```
Class A
\\Define Class A
{
[Key]     int a;
}
Class B:A
 \\Class B extends A
{
[overrides ("a", key (false)]   int b;
}
```

Sun WBEM User Manager

Sun WBEM User Manager is a software application in which you can set user privileges to specific areas, called namespaces, where classes are stored. You can also delete namespaces and create new namespaces in Sun WBEM User Manager. For information about how to use Sun WBEM User Manager, see Chapter 12.

Solaris Provider

Solaris WBEM Services includes the Solaris Provider, a program that enables the CIM Object Manager to communicate with the Solaris operating environment. The Solaris Provider is defined in a set of files created in Managed Object Format (MOF). Collectively, these files are referred to as the Solaris Schema. They extend CIM classes for the Solaris environment by providing definitions of the classes that the CIM Object Manager and the Solaris environment use to communicate.

The MOF files that make up the Solaris Schema are located in `/opt/SUNWconn/wbem/schema`. You can view these files in a text editor of your choice. `Solaris_Schema1.0.mof` is the principal schema file. It contains pointers to the other files that make up the Solaris Schema in the order in which the files are compiled at installation.

During the installation of Solaris WBEM Services, the MOF compiler compiles standard CIM 2.1 MOF files into the CIM Object Manager. After installation, these compiled classes represent the resources on your system, such as processes, application software, CPU resources, and memory. Applications can then use the API to get, set, and otherwise manipulate the managed resources on any WBEM-enabled system.

In addition, Solaris WBEM Services includes the Solaris Schema, MOF files that further describe Solaris-specific resources, such as Solaris patches and installed software packages. The Solaris Schema extends the standard CIM Schema classes.

Other vendors who extend the standard CIM Schema also build on the base classes. The benefit of using this information model to manage systems is that an application can get and set the properties for any system resource (for example, process) on any CIM system. You can use the same API to get and set properties about a process or device on a Microsoft Windows 32 system, a Solaris system, a UNIX platform, or any other CIM-compliant platform.

Installation Prerequisites

Prior to installing the Solaris WBEM Services, ensure that Sun Directory Services (SDS) version 3.1 or a compatible version is installed. SDS is used for the CIM Repository.

Shared Packages

You can install Solaris WBEM Services as a product that runs on its own, or you can install both Solaris WBEM Services and the Sun WBEM SDK to be used interactively. Installing either product involves installing the product packages. The packages are compilations of the files, interfaces, and components of each product.

Solaris WBEM Services and the Sun WBEM SDK share some of the same packages. For example, both applications require the package named `SUNWwbapi`, that contains the Client APIs.

For information about Solaris WBEM Services packages and installation instructions, see the following section, “Installing Solaris WBEM Services” on page 140. For information about Sun WBEM SDK packages and installation instructions, see Chapter 2.

Installing Solaris WBEM Services

The following table describes the packages you need to install Solaris WBEM Services.

TABLE 10–1 Solaris WBEM Services Packages

Required Packages		
Package Name	**Title**	**Description**
`SUNWwbapi`	Sun WBEM SDK - APIs	Contains the client and provider APIs and additional functionality required to run Solaris WBEM Services and the Sun WBEM SDK. This package is provided with the Sun WBEM SDK. It is required by both products.

TABLE 10–1 Solaris WBEM Services Packages *(continued)*

SUNWwbcor	Solaris WBEM Services	Contains Solaris WBEM Services components, including the MOF Compiler and the CIM Object Manager.
SUNWwbxml	Solaris WBEM Services - XML Libraries	Contains the XML libraries that enable conversion between XML and Managed Object Format (MOF).
Optional Packages		
Package Name	**Title**	**Description**
SUNWwbdoc	Solaris WBEM Services - Documentation	Contains the *WBEM Developer's Guide*, which supports both Solaris WBEM Services and the Sun WBEM SDK. Although this package is provided with Solaris WBEM Services, it can be installed optionally to support either product.
Localized Packages		
Package Name	**Title**	**Description**
SUNW*xx*wbs	Solaris WBEM Services - Localization	Contains the localized version of Solaris WBEM Services. The *xx* is replaced by the character code that represents the particular language in which the application is localized. For example, the French version of Solaris WBEM Services is packaged in SUNWfrwbs.

▼ How to Install Solaris WBEM Services

1. Become root on your system by typing the following command:

```
% su
```

2. Type the root password when you are prompted.

3. **Change directories to the location of the packages in your work environment.**

4. **At the system prompt, type the following command to obtain a list of packages:**

```
# pkgadd -d .
```

   The list of packages is displayed. You are prompted to select one or all packages.

5. **Type the number of the package you want to install.**

   - Type `1` to install the `SUNWwbapi` package. It is important to install this package first because the other packages rely on the Sun WBEM APIs.
   - Type `2` to install the `SUNWwbcor` package, which installs Solaris WBEM Services.

     When the `SUNWwbcor` package installs, the installation routine prompts you to provide the Sun Directory Services (SDS) administration password. If you have already installed SDS prior to installing Solaris WBEM Services, type the SDS password at the prompt. If you have not previously installed SDS, type a password of your choice at the prompt to set the SDS password. When you are prompted to re-enter the password, type the password again at the prompt.
   - Type `7` to install the `SUNWwbxml` package, which installs the XML Libraries.
   - (Optional): Type `3` to install the `SUNWwbdoc` package, which installs this guide.

   As each package installs, its contents are listed for you to view. When the installation is complete, you are notified with the message: `Installation of package_name was successful`.

6. **When you have finished installing the packages, type `q` to exit the package installation routine.**

7. **Type `exit` at the system prompt to exit root.**

Configuring After Installing in Solaris 7

When you install Solaris WBEM Services as part of a Solaris Easy Access Server 3.0 installation in Solaris 7, you may not be prompted to enter a password for the Sun Directory Services (SDS) administrative account. If you did not enter this password during the installation, run the `wbemconfig` script as described in the following procedure to start SDS and configure Solaris WBEM Services. The `wbemconfig` script completes the following tasks:

- Configures and starts SDS
- Sets Java environment variables
- Starts the CIM Object Manager
- Starts the MOF Compiler and compiles two MOF files:
  - `CIM_Schema21.mof` – provides the classes that make up the CIM Schema
  - `Solaris_Schema1.0.mof` – provides the classes that make up the Solaris Schema

▼ How to Configure Your Environment After Installation

1. **Become root on your system by typing the following command:**

```
% su
```

2. **Type the root password when you are prompted.**

3. **Run the `wbemconfig` script using the following command:**

```
# /opt/SUNWconn/wbem/bin/wbemconfig
```

4. **When prompted, type a password of your choice to be set as the SDS administrative account password.**

   The `wbemconfig` script runs. SDS starts followed by the CIM Object Manager. The MOF Compiler starts and compiles the CIM and Solaris Schema files.

Uninstalling Solaris WBEM Services

When you want to uninstall Solaris WBEM Services from your computer, you remove the packages. When you remove the Solaris WBEM Services packages, not all files

that make up your WBEM installation are removed. If Sun WBEM SDK is installed, none of its associated packages are removed. For information about removing Sun WBEM SDK, see “Uninstalling the Sun WBEM SDK” on page 16 in Chapter 2.

If you uninstall both the Sun WBEM SDK and Solaris WBEM Services, the LDAP schema and data files remain installed. You can remove these files, and the subdirectories that contain them, from the path `/opt/SUNWconn/ldap`. However, if you remove the LDAP data, you may encounter errors in other applications that require the data. Also, if you remove the LDAP data, you will need to re-install it if you decide to re-install the Sun WBEM SDK or Solaris WBEM Services at a later date.

▼ How to Uninstall Solaris WBEM Services

1. **Become root on your system by typing the following command:**

```
% su
```

2. **Type the root password at the Password prompt.**

3. **Type the following command at the system prompt to remove a package:**

```
# pkgrm package_name
```

   where package_name is replaced by the name of the package that you want to remove.

4. **Type** `y` **when you are prompted with the question:**
   `"Do you want to remove this package?"`

   You can remove the following packages in any order:

   - `SUNWwbcor`
   - `SUNWwbxml`

     `SUNWwbdoc`

   Be sure to remove the `SUNWwbapi` package last because all other packages rely on it.

   When a package has been removed successfully, the following message is displayed.

```
Removal of package_name was successful
```

5. **Type the `pkgrm` command at the system prompt for each package you want to remove.**

6. **Type `exit` to exit root and return to your system prompt when you have finished removing packages.**

CHAPTER 11

CIM Object Manager

When Solaris WBEM Services is installed, the CIM Object Manager starts automatically and runs continuously. It restarts automatically between system sessions or after a power outage. At times, such as if you change a provider program, you have to manually restart the CIM Object Manager.

Note - If you change a provider program, you must stop and restart the CIM Object Manager before you can use the updated provider.

This chapter describes how and when to stop and restart the CIM Object Manager. The following topics are covered.

- About CIM Object Manager
- The `init.wbem` Command
- The `cimom` Command
- Stopping the CIM Object Manager
- Restarting the CIM Object Manager
- Error Messages Generated by the CIM Object Manager

About CIM Object Manager

During the installation of Solaris WBEM Services, the CIM Object Manager starts automatically. Generally, you do not need to stop the CIM Object Manager. However, if you change a provider program, you must stop and restart the CIM Object Manager before you can use the updated provider.

Solaris WBEM Services provides one way to stop the CIM Object Manager and two ways to restart it. Use the `init.wbem` command to stop the CIM Object Manager. Use the `init.wbem` command to restart the CIM Object Manager between sessions, or use the `cimom` command to restart the CIM Object Manager on a specific host or view the version of the CIM Object Manager.

To stop a running CIM Object Manager or restart a stopped CIM Object Manager, you must be logged in as root. For information about stopping the CIM Object Manager, see "Stopping the CIM Object Manager" on page 150. For information about restarting the CIM Object Manager after it has been stopped, see "Restarting the CIM Object Manager" on page 150.

The `init.wbem` Command

The `init.wbem` restarts the CIM Object Manager between system sessions. The `init.wbem` command can also be used to stop the CIM Object Manager.

Location of the `init.wbem` Command

The `init.wbem` command is located in the following path: `/etc/init.d/init.wbem`.

Syntax of the `init.wbem` Command

The `init.wbem` command uses only two parameters:

Parameter	Description
stop	Enables you to stop the CIM Object Manager
start	Enables you to restart the CIM Object Manager

The syntax of the `init.wbem` command is:

- `init.wbem stop`
- `init.wbem start`

The cimom Command

Use the cimom command to start the CIM Object Manager when you want to obtain or specify additional information, such as the host that contains a CIM Repository in which you want your objects to be stored. The cimom command uses parameters that enable you to specify characteristics of the CIM Object Manager.

Location of the cimom Command

By default, the cimom command is located in the following path:
/opt/SUNWconn/wbem/bin/

Syntax of the cimom Command

The cimom command uses the following three parameters.

Parameter	Description
-help	Causes a man page to display with information about the cimom command and parameters.
-s*database_server*	Enables you to specify a server installed with a CIM Repository. This parameter provides control over where your CIM objects are stored.
-version	Causes the build version of the CIM Object Manager to be displayed.

The syntax of the cimom command is: cimom *[parameter] [modifier]*

where [parameter] is any of the parameters shown in the preceding table, and [modifier] is the additional information that is required by a parameter. For example in the command cimom *-s hopskotch*, *-s* is the *[parameter]* and *hopskotch* is the *[modifier]* that indicates the name of the server on which the CIM Repository is located.

Stopping the CIM Object Manager

Use the following procedure to stop the CIM Object Manager.

▼ How to Stop the CIM Object Manager

1. **Become root on your system by typing the following command at the system prompt:**

```
% su
```

2. **Type the root password when you are prompted.**

3. **Change directories to the location of the `init.wbem` command by typing the following command:**

```
# cd /etc/init.d/
```

4. **Stop the CIM Object Manager by typing the following command:**

```
# ./init.wbem stop
```

The CIM Object Manager stops.

Restarting the CIM Object Manager

You can restart the CIM Object Manager using the `init.wbem` command or the `cimom` command, depending on how you want to restart. If you want to restart the CIM Object Manager on the default host, use the `init.wbem` command. If you want to restart the CIM Object Manager on another host, use the `cimom` command. Using the `cimom` command, you can also view the version of the CIM Object Manager.

▼ How to Restart the CIM Object Manager Using Default Settings

1. **Become root on your system by typing the following command at the system prompt:**

```
% su
```

2. **Type the root password when you are prompted.**

3. **Change directories to the location of the `init.wbem` command by typing the following command:**

```
# cd /etc/init.d/
```

4. **Restart the CIM Object Manager by typing the following command:**

```
# ./init.wbem start
```

The CIM Object Manager starts.

▼ How to Restart the CIM Object Manager and Specify a Host

1. **Become root on your system by typing the following command at the system prompt:**

```
% su
```

2. **Type the root password when you are prompted.**

3. **Change directories to the location of the `cimom` command by typing the following command:**

```
# cd /opt/SUNWconn/wbem/bin
```

4. **Restart the CIM Object Manager and specify a host by typing the following command:**

```
# cimom -s server_name
```

where *server_name* is the name of a specific host running a CIM Object Manager Repository.

The CIM Object Manager resumes running.

Error Messages Generated by the CIM Object Manager

The CIM Object Manager generates error messages to indicate incorrect MOF syntax and semantics. The same error messages are generated by the MOF Compiler and the CIM Workshop. To view the error messages and their meanings, see Chapter 9.

CHAPTER 12

Administering Security

This chapter describes the security features enforced by the CIM Object Manager, including the following topics:

- Overview
- Using the Sun WBEM User Manager to Set Access Control
- Using the APIs to Set Access Control
- Error Messages

Overview

The CIM Object Manager validates a user's login information for the machine on which the CIM Object Manager is running. A validated user is granted some form of controlled access to the entire Common Information Model (CIM) Schema. The CIM Object Manager does not provide security for system resources such as individual classes and instances. However, the CIM Object Manager does allow control of global permissions on namespace and access control on a per-user basis.

All security-related information is represented by instances of security classes located in the `root\Security` namespace and must remain there permanently.

The following security features protect access to CIM objects on a WBEM-enabled system:

- Authentication - The process of verifying the identity of a user, device, or other entity in a computer system, often as a prerequisite to allowing access to the resources in a system.
- Authorization – The granting to a user, program, or process the right of access.

- Replay protection – A client cannot copy another client's last message sent to a CIM Object Manager. The CIM Object Manager uses the client keys to guarantee that all subsequent communication in the client-server session is with the same client that initiated the session and participated in the client-server authentication.

  The CIM Object Manager protects against a client picking up and sending another client's message to the server by validating digitally signed secret session keys. The CIM Object Manager will not accept an identical byte stream from a client without a valid secret session key.

- Digital signature – The CIM Object Manager uses Java digital signature classes to digitally sign the clients response to the server, however it does not digitally sign the server's response to a client.

Authentication

When a user logs in and enters a user name and password, the client encrypts the password and sends the encrypted password to the CIM Object Manager. When the user is authenticated, the CIM Object Manager sets up a client session. All subsequent operations occur within that secure client session.

Authorization

The CIM Object Manager creates two user accounts:

- `wbemadmin` – The administrative account used to access LDAP schema in the CIM Object Manager Repository. The `wbemadmin` account is created and its password is set during installation.
- `guest` – The default account used when no user name is specified during login.

Once the CIM Object Manager has authenticated the user's identity, that identity can be used to verify whether the user should be allowed to execute the application or any of its tasks. The CIM Object Manager supports capability-based authorization, which allows an administrator to assign read and write access to specific users. These authorizations are added to existing Solaris user accounts.

Note - We do not recommend logging in as root because successful login to the root account depends on how name services (for example, DNS, NIS, or NIS+) are set up on your system.

Using the Sun WBEM User Manager to Set Access Control

The Sun WBEM User Manager allows administrators to add and delete authorized users and to set their access privileges. Use this application to manage user authentication and access to CIM objects on a WBEM-enabled system. A user must have a Solaris user account.

You can set access privileges on individual users, on namespaces, or on both. When you add a user, you select a namespace. This action grants the user read access to CIM objects in the selected namespace.

Administrators are users who are logged in to the WBEM administrative account, `wbemadmin`. Administrators can set the following types of access to CIM objects:

- Read Only – Allows read-only access to CIM Schema objects. Users with this privilege can retrieve instances and classes, but cannot create, delete, or modify CIM objects.
- Read / Write – Allows full read / write / delete access to all CIM classes and instances.
- Write – Allows write and delete, but not read access to all CIM classes, and instances.
- None – Allows no access to CIM classes and instances.

▼ How to Start Sun WBEM User Manager

1. **In a command window, type the command:**

```
% /opt/SUNWconn/wbem/bin/cimadmin
```

   The Sun WBEM User Manager is started. The User Manager and Login dialog boxes are displayed at the same time. The Login dialog box shows the name of the current host. Context-help information is available on the fields in the dialog box.

2. **In the Login dialog box, do the following:**
   - In the Host Name field, type the name of a host running the CIM Object Manager.

- In the User Name field, type `wbemadmin`. You must log in to the administrative account to administer WBEM user accounts.
- In the Password field, type the password for the `wbemadmin` account.

3. **Click OK.**

   The User Manager dialog box opens with a list of users and their access rights to WBEM objects within the namespaces on the current host.

▼ How to Grant Default Access Rights to a User

1. **Start Sun WBEM User Manager.**
2. **In the Users Access portion of the dialog box, click Add.**

   A dialog box opens that lists the available namespaces.
3. **Type the name of a Solaris user account in the User Name text entry field.**
4. **Select a namespace from the listed namespaces.**
5. **Click OK.**

   This action grants this user read access to CIM objects in the selected namespace. The user is added to the User Manager dialog box.
6. **Click OK again to close the User Manager dialog box.**

▼ How to Change Access Rights for a User

1. **Start Sun WBEM User Manager.**
2. **Select the user whose access rights you want to change.**
3. **To grant the user read-only access, click the Read check box. To grant the user write access, click the Write check box.**
4. **Click OK.**

▼ How to Remove Access Rights for a User

1. **Start Sun WBEM User Manager.**

2. **In the Users Access portion of the dialog box, select the user name for which you want to remove access rights.**

3. **Click Delete to delete the user's access rights to the namespace.**

   A confirmation dialog box asks you to confirm your decision to delete the user's access rights. Click OK to confirm.

4. **Click OK again to close the User Manager dialog box.**

▼ How to Set Access Rights for a Namespace

1. **Start Sun WBEM User Manager.**

2. **In the Namespace Access portion of the dialog box, click Add.**

   A dialog box opens that lists the available namespaces.

3. **Select the namespace for which you want to set access rights.**

   By default, users have read-only access to a namespace.

4. **To allow no access to the namespace, make sure the Read and Write check boxes are not selected. To allow write access, click the Write check box. To allow read access, click the Read check box.**

5. **Click OK to close the User Manager dialog box.**

▼ How to Remove Access Rights for a Namespace

1. **Start Sun WBEM User Manager.**

2. **In the Namespace Access portion of the dialog box, select the namespace for which you want to remove access control, and then click Delete.**

   Access control is removed from the namespace, and the namespace is removed from the list of namespaces on the User Manager dialog box.

3. **Click OK to close the User Manager dialog box.**

Using the APIs to Set Access Control

You can use the Sun WBEM SDK APIs to set access control on a namespace or on a per-user basis. During installation, the MOF compiler compiles the security classes defined in the `Solaris_Acl1.0.mof` file into the `/root/Security` namespace. The `Solaris_Acl1.0.mof` file defines the following classes:

- `Solaris_Acl` – Base class for Solaris Access Control Lists (ACL). This class defines the string property *capability* and sets its default value to `r` (read only).
- `Solaris_UserAcl` – Represents the access control that a user has to the CIM objects within the specified namespace.
- `Solaris_NamespaceAcl` – Represents the access control on a namespace.

You can set access control on individual users to the CIM objects within a namespace by creating an instance of the `Solaris_UserACL` class and then using the APIs to change the access rights for that instance. Similarly, you can set access control on namespaces by creating an instance of the `Solaris_NameSpaceACL` class and then using APIs, such as the `setInstance` method, to set the access rights for that instance.

An effective way to combine the use of these two classes is to first use the `Solaris_NameSpaceACL` class to restrict access to all users to the objects in a namespace. Then use the `Solaris_UserACL` class to grant selected users access to the namespace.

Note - Access Control Lists (ACL) are governed by a standard being developed by the DMTF. Although the Solaris ACL schema are currently CIM-compliant, they will need to change when the DMTF finalizes the ACL standard. Programs you write using the Solaris ACL schema classes are subject to that risk.

The `Solaris_UserAcl` Class

The `Solaris_UserAcl` class extends the `Solaris_Acl` base class, from which it inherits the string property *capability* with a default value `r` (read only).

You can set the *capability* property to any of the following values for access privileges.

Access Right	Description
r	Read
rw	Read and Write
w	Write
none	No access

The Solaris_UserAcl class defines the following two key properties. Only one instance of the namespace-username ACL pair can exist in a namespace.

Property	Data Type	Purpose
nspace	string	Identifies the namespace to which this ACL applies.
username	string	Identifies the user to which this ACL applies.

▼ How to Set Access Control on a User

1. **Create an instance of the** Solaris_UserAcl **class. For example:**

```
// Get the Solaris_UserAcl class
cimclass = cc.getClass(newCIMObjectPath("Solaris_UserAcl");

// Create a new instance of the Solaris_UserAcl class
ci = cimclass.newInstance();
```

2. **Set the *capability* property to the desired access rights. For example:**

```
/* Change the access rights (capability) to read/write
for user Guest on objects in the root\molly namespace.
ci.updatePropertyValue("capability",new CIMValue("rw"));
ci.updatePropertyValue("nspace",new CIMValue("root\molly"));
ci.updatePropertyValue("username",new CIMValue("guest"));
```

3. **Update the instance. For example:**

```
// Pass the updated instance to the CIM Object Manager
cc.setInstance(new CIMObjectPath(), ci);
```

The `Solaris_NamespaceAcl` Class

The `Solaris_NamespaceAcl` extends the `Solaris_Acl` base class, from which it inherits the string property *capability* with a default value `r` (read-only for `GUEST` and all users). The `Solaris_NamespaceAcl` class defines the following key property.

Property	Data Type	Purpose
nspace	string	Identifies the namespace to which this access control list applies. Only one instance of the namespace ACL can exist in a namespace.

▼ How to Set Access Control on a Namespace

1. **Create an instance of the `Solaris_namespaceAcl` class. For example:**

```
// Get the Solaris_namespaceAcl class
cimclass = cc.getClass(newCIMObjectPath("Solaris_namespaceAcl");

// Create a new instance of the Solaris_namespaceAcl class
ci = cimclass.newInstance();
```

2. **Set the *capability* property to the desired access rights. For example:**

```
/* Change the access rights (capability) to read/write
to the root\molly namespace. */
ci.updatePropertyValue("capability",new CIMValue("rw"));
ci.updatePropertyValue("nspace",new CIMValue("root\molly"));
```

3. **Update the instance. For example:**

```
// Pass the updated instance to the CIM Object Manager
cc.setInstance(new CIMObjectPath(),ci);
```

Error Messages

For a description of error messages, see Chapter 9.

CHAPTER 13

Logging Events

Logging is a service that enables WBEM administrators to track uncommon events to determine how they occurred. This chapter covers the following topics:

- About Logging
- Log Files
- Log Classes
- Viewing Log Data
- Using the APIs to Enable Logging

About Logging

The logging service records all actions completed by Solaris WBEM Services or Sun WBEM SDK components. Informational and error content can be recorded to a log. For example, if a user disables a serial port, this information can be logged automatically by a serial port provider. Or, if a system error or other failure occurs, the WBEM administrator can check the log record to trace the cause of the occurrence.

All Sun WBEM SDK and Solaris WBEM Services components, applications, and providers start logging automatically, in response to events. For example, the CIM Object Manager automatically logs events after is installed and started.

You can set up logging for applications and providers that you develop for the WBEM environment. For information, see “Using the APIs to Enable Logging” on page 165. You can also view log data in CIM WorkShop for administration purposes or to debug the logging functionality that you develop for applications.

Log Files

When you set up an application or a provider to log events, its events are recorded in log files. All log records are stored in the path: `/var/opt/SUNWconn/wbem/log/`. Log files use the following naming convention:

```
wbem_log.#
```

where # is a number appended to indicate the version of the log file. A log file appended with a `.1`, such as `wbem_log.1`, is the most recently-saved version. A log file appended with a `.2` is the next oldest version. Larger file extensions, for example, `wbem_log.16`, indicate older versions of the file. Previous versions of the log file and the most recent version co-exist as an archive in `/var/opt/SUNWconn/wbem/log`.

Log files are renamed with a `.1` file extension, saved, and archived when one of the following two conditions are met:

- The current file reaches the file size limit specified by the `Solaris_LogServiceProperties` class

  For information about how the properties of the `Solaris_LogServiceProperties` class control how a log file is used, see "Log File Rules" in "Log File Rules" on page 162.

- The `clearLog()` method of the `Solaris_LogService` class is invoked on the current log file

  For information about the `Solaris_LogService` class and its methods, see "Solaris_LogService" on page 164.

Log File Rules

The `Solaris_LogServiceProperties` class is defined in `Solaris_Core1.0.mof`. The `Solaris_LogServiceProperties` class has properties that control the following attributes of a log file:

- Directory where the log file is written
- Name of the log file
- Date the log file was created
- Size allowed for a log file before it is renamed with a `.1` file extension, saved, and archived in `/var/opt/SUNWconn/wbem/log`
- Number of log files you can have in the archive
- Ability to write log data to SysLog, the default logging system of the Solaris operating environment

When you want to specify any of these attributes for an application that writes data to a log file, you create a new instance of `Solaris_LogServiceProperties` and set the values of its associated properties.

Log File Format

The logging service provides three general types of log files: application logs, system logs, and security logs. Log records may be informational, or may record data derived from errors or warnings. A standard set of fields are defined for the data that can be presented in logs; however, logs do not necessarily use all fields. For example, an informational log may provide a brief message describing an event. An error log may provide a more detailed message.

Some log data fields identify data in the CIM Repository. These fields are properties flagged with a read-only key qualifier in the `Solaris_LogRecord` class. You cannot set the values of these fields. You can set the values of any of the following fields in your log files:

- Category – type of log file.
- Severity – Severity of conditions that caused data to be written to a log file.
- AppName – Name of the application from which the data was obtained.
- UserName – Name of the individual who was using the application when log data was generated.
- ClientMachineName – Name of the computer on which an incident occurred that generated log data.
- ServerMachineName – Name of the server on which an incident occurred that generated log data.
- SummaryMessage – Brief message describing the occurrence.
- DetailedMessage – Detailed message describing the occurrence.
- Data – Context information that applications and providers can present to interpret a log message.

Log Classes

Logging uses two Solaris Schema classes: `Solaris_LogRecord` and `Solaris_LogService`.

Solaris_LogRecord

`Solaris_LogRecord` is defined in `Solaris_Core1.0.mof` to model an entry in a log file. When an application or provider calls the `Solaris_LogRecord` class in response to an event, the `Solaris_LogRecord` class causes all data generated by the event to be written to a log file. To see the definition of the `Solaris_LogRecord` class as part of the Solaris Provider, view the `Solaris_Core1.0.mof` file in a text editor of your choice. The `Solaris_Core1.0.mof` file is located in `/opt/SUNWconn/wbem/schema`.

`Solaris_LogRecord` uses a vector of properties and key qualifiers to specify attributes of the events, system, user, and application or provider that generate data. Read-only qualifier values are generated transparently for use between the application and the CIM Repository. For example, the value RecordID uniquely identifies the log entry but is not displayed as part of the log format when you view generated data.

You can set the values of writeable qualifier values. For example, you can set the qualifier values of properties such as `ClientMachineName` and `ServerMachineName` which identify the system on which an event occurs.

Solaris_LogService

The `Solaris_LogService` class controls the operation of the logging service and defines the ways in which log data is handled. This class has a set of methods that an application can use to distribute data about a particular event to the CIM Object Manager from the issuing application. The data becomes a trigger that generates a response from the CIM Object Manager, such as a retrieval of data from the CIM Repository.

The `Solaris_LogService` class uses the following methods:

- `clearLog` – Renames, saves, and archives a current log file or deletes an archived log file
- `getNumRecords` – Returns the number of lines of data recorded in a particular log file
- `listLogFiles` – Returns a list of all log files stored in `/var/opt/SUNWconn/wbem/log`
- `getCurrentLogFileName` – Returns the name of the most recent log file
- `getNumLogFiles` – Returns the number of log files stored in `/var/opt/SUNWconn/wbem/log`
- `getLogFileSize` – Returns the size, in megabytes, of a particular log file
- `getSyslogSwitch` – Enables log data to be sent to SysLog, the logging service of the Solaris operating environment

- `getLogStorageName` – Returns the name of the host computer or device where log files are stored
- `getLogFileDir` – Returns the path and name of the directory where log files are stored
- `setProperties` – Enables you to set logging properties

You can view the definition of `Solaris_LogService` in the `Solaris_Core1.0.mof` file by opening the file in a text editor of your choice. The `Solaris_Core1.0.mof` file is located in `/opt/SUNWconn/wbem/schema`.

Viewing Log Data

If your application has been enabled to log data in response to events, you can view the generated data in CIM Workshop.

▼ How to View Log Data

1. **In CIM Workshop, select `solaris_logrecord` in the class inheritance tree.**

2. **Click Action->Instances.**

3. **In the left side of the Instances for `Solaris_LogRecord` window, click the instance that was generated by your application.**

   In the right side of the Instances for `Solaris_LogRecord` window, the fields of the log file are displayed as properties. Values for each property are provided in the Values column.

Using the APIs to Enable Logging

Currently, you can view log file content in CIM Workshop. However, you can develop your own log viewer if you prefer to view log files in a customized manner. You can use the logging application programming interfaces (APIs) to develop a log viewer. The APIs enable you to write data from an application to a log file or read data from a log file to your log viewer.

Writing Data to a Log File

Enabling an application to write data to a log file involves the following main tasks:

- Creating a new instance of the `Solaris_LogRecord` class
- Specifying the properties that will be written to the log file and setting values for the property qualifiers
- Setting the new instance and properties to print

▼ How to Create an Instance of Solaris_LogRecord to Write Data

1. Import all necessary java.rmi classes.

CODE EXAMPLE 13–1 Importing Classes

```
import java.rmi.*;
import com.sun.wbem.client.CIMClient;
import com.sun.wbem.cim.CIMInstance;
import com.sun.wbem.cim.CIMValue;
import com.sun.wbem.cim.CIMProperty;
import com.sun.wbem.cim.CIMNameSpace;
import com.sun.wbem.cim.CIMObjectPath;
import com.sun.wbem.cim.CIMClass;
import com.sun.wbem.cim.CIMException;
import com.sun.wbem.solarisprovider.*;
import java.util.*;
import java.util.Enumeration;
```

2. Declare the public class `CreateLog` and the following values:

- `CIMClient` value
- `CIMObjectPath` value

- `CIMNameSpace` value

CODE EXAMPLE 13–2 Declaring the CreateLog Class and Values

```
public class CreateLog {
    public static void main(String args[]) throws CIMException {

 if ( args.length != 3) {
     System.out.println("Usage: CreateLog host username password");
     System.exit(1);
 }

 CIMClient cc = null;
 CIMObjectPath cop = null;
 try {
     CIMNameSpace cns = new CIMNameSpace(args[0]);
     cc = new CIMClient(cns, args[1], args[2]);
```

3. Specify the vector of properties to be returned. Set values for the properties of the qualifiers.

CODE EXAMPLE 13–3 Specifying the Vector of Properties and their Values

```
Vector keys = new Vector();
CIMProperty logsvcKey = new CIMProperty("RecordID");
logsvcKey.setValue(new CIMValue(new Integer(0)));
keys.addElement(logsvcKey);
logsvcKey = new CIMProperty("RecordHashCode");
logsvcKey.setValue(new CIMValue(new Integer(0)));
keys.addElement(logsvcKey);
logsvcKey = new CIMProperty("Filename");
logsvcKey.setValue(new CIMValue("some_file"));
keys.addElement(logsvcKey);
logsvcKey = new CIMProperty("category");
logsvcKey.setValue(new CIMValue(new Integer(2)));
keys.addElement(logsvcKey);
logsvcKey = new CIMProperty("severity");
logsvcKey.setValue(new CIMValue(new Integer(2)));
keys.addElement(logsvcKey);
logsvcKey = new CIMProperty("AppName");
logsvcKey.setValue(new CIMValue("SomeApp"));
keys.addElement(logsvcKey);
logsvcKey = new CIMProperty("UserName");
logsvcKey.setValue(new CIMValue("molly"));
keys.addElement(logsvcKey);
logsvcKey = new CIMProperty("ClientMachineName");
```

(continued)

(Continuation)

```
logsvcKey.setValue(new CIMValue("dragonfly"));
keys.addElement(logsvcKey);
logsvcKey = new CIMProperty("ServerMachineName");
logsvcKey.setValue(new CIMValue("spider"));
keys.addElement(logsvcKey);
logsvcKey = new CIMProperty("SummaryMessage");
logsvcKey.setValue(new CIMValue("brief_description"));
keys.addElement(logsvcKey);
logsvcKey = new CIMProperty("DetailedMessage");
logsvcKey.setValue(new CIMValue("detailed_description"));
keys.addElement(logsvcKey);
logsvcKey = new CIMProperty("data");
logsvcKey.setValue(new CIMValue("0xfe 0x45 0xae 0xda"));
 keys.addElement(logsvcKey);
logsvcKey = new CIMProperty("SyslogFlag");
logsvcKey.setValue(new CIMValue(new Boolean(true)));
keys.addElement(logsvcKey);
```

4. Declare the new instance of the `CIMObject Path` for the log record.

CODE EXAMPLE 13–4 Declaring the New Instance of CIMObjectPath

```
CIMObjectPath logreccop = new CIMObjectPath("Solaris_LogRecord", keys);
```

5. Declare the new instance of `Solaris_LogRecord`. Set vector of properties to write to a file.

CODE EXAMPLE 13–5 Setting the Instance and Properties

```
CIMInstance ci = new CIMInstance();
  ci.setClassName("Solaris_LogRecord");
  ci.setProperties(keys);
  //System.out.println(ci.toString());
  cc.setInstance(logreccop,ci);
 }
 catch (Exception e) {
     System.out.println("Exception: "+e);
  e.printStackTrace();
 }
```

6. Close the session after data has been written to the log file.

CODE EXAMPLE 13–6 Closing the Session

```
 // close session.
 if(cc != null) {
     cc.close();
 }
    }
}
```

Reading Data from a Log File

Enabling an application to read data from a log file to a log viewer involves the following tasks:

- Enumerating instances of the `Solaris_LogRecord` class
- Getting the desired instance
- Printing properties of the instance to an output device, typically a user interface

▼ How to Get an Instance of Solaris_LogRecord and Read Data

1. Import all necessary java.rmi classes.

CODE EXAMPLE 13–7 Importing Classes

```
import java.rmi.*;
import com.sun.wbem.client.CIMClient;
import com.sun.wbem.cim.CIMInstance;
import com.sun.wbem.cim.CIMValue;
import com.sun.wbem.cim.CIMProperty;
import com.sun.wbem.cim.CIMNameSpace;
import com.sun.wbem.cim.CIMObjectPath;
import com.sun.wbem.cim.CIMClass;
import com.sun.wbem.cim.CIMException;
import com.sun.wbem.solarisprovider.*;
import java.util.*;
```

(continued)

(Continuation)

```
import java.util.Enumeration;
```

2. Declare the class `ReadLog`**.**

CODE EXAMPLE 13–8 Declaring the ReadLog Class

```
public class ReadLog
    {
    public static void main(String args[]) throws
    CIMException
    {
    if ( args.length != 3)
    {
    System.out.println("Usage: ReadLog host username
    password");
     System.exit(1);
```

3. Set client, object path, and namespace values of the ReadLog class.

CODE EXAMPLE 13–9

```
 }
CIMClient cc = null;
 CIMObjectPath cop = null;
try {      CIMNameSpace cns = new CIMNameSpace(args[0]);
cc = new CIMClient(cns, args[1], args[2]);
cop = new CIMObjectPath("Solaris_LogRecord");
```

4. Enumerate instances of `Solaris_LogRecord`**.**

CODE EXAMPLE 13–10 Enumerating Instances

```
 Enumeration e = cc.enumInstances(cop, true);
  for (; e.hasMoreElements(); ) {
```

5. Send property values to an output device.

CODE EXAMPLE 13–11 Sending Property Values

```
 System.out.println("------------------------
---------");
   CIMObjectPath op = (CIMObjectPath)e.nextElement();
   CIMInstance ci = cc.getInstance(op);
   System.out.println("Record ID : " +
      (((Long)ci.getProperty("RecordID").getValue().
      getValue()).longValue()));
   System.out.println("Log filename : " +
      ((String)ci.getProperty("FileName").getValue().
      getValue()));
   int categ = (((Integer)ci.getProperty("category").
      getValue().getValue()).intValue());
   if (categ == 0)
    System.out.println("Category : Application Log");
   else if (categ == 1)
    System.out.println("Category : Security Log");
   else if (categ == 2)
    System.out.println("Category : System Log");
   int severity = (((Integer)ci.getProperty
      ("severity").getValue().getValue()).intValue());
   if (severity == 0)
    System.out.println("Severity : Informational");
   else if (severity == 1)
    System.out.println("Severity : Warning Log!");
   else if (severity == 2)
    System.out.println("Severity : Error!!");
   System.out.println("Log Record written by :" +
      ((String)ci.getProperty("AppName").getValue().
      getValue()));
   System.out.println("User : " + ((String)ci.
      getProperty("UserName").getValue().getValue()));
   System.out.println("Client Machine : " + ((String)ci.
      getProperty("ClientMachineName").getValue().getValue
      ()));
   System.out.println("Server Machine : " + ((String)ci.
      getProperty("ServerMachineName").getValue().getValue
      ()));
   System.out.println("Summary Message : " + ((String)
      ci.getProperty("SummaryMessage").getValue().getValue
      ()));
   System.out.println("Detailed Message : " + ((String)
```

(continued)

(Continuation)

```
      ci.getProperty("DetailedMessage").getValue().getValue
      ()));
    System.out.println("Additional data : " + ((String)
      ci.getProperty("data").getValue().getValue()));
    boolean syslogflag =
     ((Boolean)ci.getProperty("syslogflag").getValue().
      getValue()).booleanValue();
     if (syslogflag == true) {
    System.out.println("Record was written to syslog as
      well");
    } else {
    System.out.println("Record was not written to
      syslog");
    }
    System.out.println("----------------------------
      ----");
  }
```

6. Return an error message to the user if an error condition occurs.

CODE EXAMPLE 13–12 Returning an Error Message

```
}
 catch (Exception e) {
     System.out.println("Exception: "+e);
  e.printStackTrace();
 }
```

7. Close the session when the data has been read from the file.

CODE EXAMPLE 13–13 Closing the Session

```
 // close session.
 if(cc != null) {
     cc.close();
 }
}
}
```

APPENDIX **A**

Common Information Model (CIM) Terms and Concepts

CIM Concepts

The following sections describe basic CIM terms and concepts that are essential to understanding how network entities and management functions are described and related within the context of CIM. For more detailed information about the Common Information Model and object-oriented modeling practices, including how to model your own schema, refer to `the CIM Tutorial` provided by the Distributed Management Task Force.

Object-Oriented Modeling

CIM uses the principles of Object-Oriented Modeling, a way to represent an object, entity, concept, or function that has a physical or logical existence. The goal of Object-Oriented Modeling is to set a representation of a physical entity into a framework, or model, to express the qualities and functions of the entity and its relationships with other entities. In the context of CIM, Object-Oriented Modeling is used to model hardware and software elements.

Uniform Modeling Language

Models are expressed in the form of visual representation and language. CIM conventions for rendering the model are based on the diagrammatic concepts of Uniform Modeling Language UML. UML uses shapes to represent physical entities

and lines to represent relationships. For example, in UML, classes are represented as rectangles. Each rectangle contains the name of the class it represents. A line between two rectangle represents a relationship between the two. A line that forks to join two classes to a higher-level class represents an association.

CIM diagrams add color to the diagrams to further express relationships:

- Red lines→Associations
- Blue lines→Inheritance relationships
- Green lines→Aggregation

CIM Terms

The following terms are innate to the CIM Schema.

Schema

The terms model, schema, and framework are synonymous. Each is an abstract representation of an entity that has a physical or logical existence. In CIM, a schema is a named collection of classes used for class naming and administration. Within a schema, classes and their subclasses are represented hierarchically using the following syntax: `Schemaname_classname.propertyname`. Each class name in a schema must be unique. Sun WBEM includes a Solaris Schema. It contains all classes specific to the Solaris extension to CIM.

Class and Instance

In WBEM, a class is a collection of objects that represents the most basic unit of management. For example, in Sun WBEM, the three main functional classes include `CIMClass`, `CIMProperty`, and `CIMInstance`.

Abstractly, classes are used to create managed objects. Class characteristics are inherited by the child objects, or instances, that are created from a class. For example, using `CIMClass`, you can create an instance, `CIMClass (Solaris_Computer_System)`.

This instance of `CIMClass` answers the question, "What is the computer system?" The value of the instance is `Solaris_Computer_System`. All instances of the same class type are created from the same class template. In the example, the name of the computer system provides a template to create managed objects of the type `Computer_System`.

Classes can be static or dynamic. Instances of static classes are stored by the CIM Object Manager and can be retrieved from the CIM Repository when a request is made. Instances of dynamic classes—classes containing data that changes regularly, such as system usage—are created by provider applications as the data changes.

Custom Classes: Extensions to CIM

For extensions to CIM, custom classes can be developed to support managed objects that are specific to their managed environment. The CIM Object Manager API provides new classes to extend CIM for the Solaris operating environment.

Property

A property defines a characteristic of a class. For example, using the `CIMProperty` class, you can define a key as a property of a particular CIM class. Values of properties can be passed back from the CIM Object Manager as a string or as a vector for a range of properties. Each property has a unique name and only one domain—the class that owns the property. A property of a given class can be overridden by a property of its subclass.

In Sun WBEM, an example of a property is the `CIMProperty`, which denotes the properties of a `CIMClass`.

Method

Like properties, methods belong to the class that owns them. A method is an action the objects of a given class are programmed to complete. For example, the method `public String getName()` returns the name of an instance as a concatenation of its keys and their values. Collectively, these actions describe the behavior of the class. Methods can belong only to the class that owns them. Within the context of a class, each method must have a unique name. A method of a given class can be overridden by a method of its subclass.

New classes inherit the definition of the method from the superclass, but not the implemented method. The definition of the method, indicated by a qualifier, serves as a placeholder in which a new implemented method can be provided. The CIM Object Manager checks for methods by starting from the lowest-level class and moving up the tree to the root class searching for a qualifier type that indicates a method.

Domain

Properties and methods are declared within a class. The class that owns the property or method is referred to as the domain of the property or method.

Qualifier and Flavor

A CIM qualifier is a modifier used to characterize CIM classes, properties, methods, and parameters. Qualifiers have unique attributes, including Name, Type, and Value, that are inherited by new classes.

Indication

An indication, an object and a type of class, is created as a result of the occurrence of an event. Indications can be arranged in a type hierarchy. Indications may have properties, methods, and triggers. Triggers are system operations, such as a change made to an existing class, or events that result in the creation of new instances of an indication.

Association

An association is a class that represents a relationship between two or more classes. Associations enable the creation of multiple relationship instances for a given class. System components can be related in many different ways, and associations provide a way of representing the relationships of these components.

Because of the way associations are defined, it is possible to establish a relationship between classes without affecting any of the related classes. The addition of an association does not affect the interface of the related classes. Only associations can have references.

Reference and Range

A reference is a type of property that defines the roles of objects involved in an association. The reference specifies the role name of the class in the context of the association. The domain of a reference is an association. The range of a reference is a character string that indicates the reference type.

Override

The override relationship is used to indicate the substitution of a property or method inherited from a subclass for a property or method inherited from the superclass. In CIM, guidelines determine what qualifiers of properties and methods can be overridden. For example, if the qualifier type of a class is flagged as a key, then the key cannot be overridden, because CIM guidelines specify that a key property cannot be overridden.

Core Model Concepts

The following sections provide descriptive information about the Core Model of CIM.

System Aspects of the Core Model

The Core Model provides classes and associations you can use to develop applications in which systems and their functions are represented as managed objects. These classes and associations embody the characteristics unique to all elements that comprise a system: physical and logical elements. Physical characteristics refer to the qualities of occupying space and conforming to the elementary laws of physics. Logical characteristics represent abstractions used to manage and coordinate aspects of the physical environment, such as system state or the capabilities of a system.

In the Core Model, logical elements can include the following.

TABLE A–1 Core Model Elements

Element Name	Description
Systems	A grouping of other logical elements. Because systems are themselves logical elements, a system can be composed of other systems.
Network Components	Classes that provide a topological view of a network.
Services and Access Points	Provide a mechanism for organizing the structures that provide access to the capabilities of a system.
Devices	An abstraction or emulation of a hardware entity, that may or may not be realized in physical hardware.

The following sections describe the classes and associations provided by the Core Model to emulate the qualities of systems.

System Classes Provided by the Core Model

The following table lists the classes that represent system aspects of the Core schema. The instances of these classes will most often belong to the descendents of the objects contained within the class.

TABLE A–2 Core Model System Classes

Class Name	Description	Example
`Managed System Element`	Base class for the system element hierarchy. Any distinguishable component of a system is a candidate for inclusion in this class.	Software components, such as files; and devices, such as disk drives and controllers, and physical components, such as chips and cards.
`Logical Element`	Base class for all the components of the system that represent abstract system components	Profiles, processes, or system capabilities in the form of logical devices.
`System`	Logical Element that aggregates an enumerable set of ManagedSystemElements. The aggregation operates as a functional whole. Within any particular subclass of System, there is a well-defined list of Managed System Element classes, whose instances must be aggregated.	Local Area Network, Wide Area Network, subnet, intranet
`Service`	Logical Element that contains the information necessary to represent and manage the functionality provided by a Device and/or SoftwareFeature. A Service is a general-purpose object to configure and manage the implementation of functionality. It is not the functionality itself.	Printer, modem, fax machine

System Associations Provided by the Core Model

Associations are classes that define the relationships shared by other classes. Association classes are flagged with an ASSOCIATION qualifer that denotes the purpose of the class. An association class must have at least two references, the names of the classes that share a particular relationship. Instances of an association always belong to the association class.

Associations can have the following types of relationships:

- One to one
- One to many
- One to zero
- Aggregation, such as a containment relationship between a system and its parts

Associations express the relationship between a system and the managed elements that make up the system. Two broad types of associations are used to define the relationships between classes:

The CIM Schema defines two basic types of associations:

- Component associations, which indicate that one class is part of another
- Dependency associations, which indicate that a class cannot function or exist without another class

These association types are abstract, which means that association classes do not have instances alone. Instances must belong to one of their descendent classes.

Component Associations

Component associations express the relationship between the parts of a system and the system itself. Component associations describe what elements make up a system. Abstract classes that express component associations are used to create concrete associations of this type in descendent classes. The descendent concrete associations answer the question: "What composition relationships does the component, or class, have with other components?"

In its most specialized role, the component association expresses the relationship between a system and its logical and physical parts.

Dependency Associations

Dependency associations establish the relationships between objects that rely on one another. The Core Model provides for the following types of dependencies:

- Functional—the dependent object cannot function without the object on which it depends

- Existence—the dependent object cannot exist without the object on which it depends

The following types of dependencies are included in the Core Model.

TABLE A–3 Core Model Dependencies

Dependency Association	Description
`HostedService`	An association between a Service and the System on which the functionality resides. The cardinality of this association is one-to-many. A System may host many Services. Services are weak with respect to their hosting System. Generally speaking, a Service is hosted on the System where the LogicalDevices or SoftwareFeatures that implement the Service are located. The model does not represent Services hosted across multiple systems. This is modeled as an ApplicationSystem that acts as an aggregation point for Services that are each located on a single host.
`HostedAccessPoint`	An association between a ServiceAccessPoint (SAP) and the System on which it is provided. The cardinality of this association is one-to-many and is weak with respect to the System. Each System may host many SAPs. A feature of the model is that the access point of a service can be located on the same or a different host from the system to which the service provides access. This allows the model to depict both distributed systems (an ApplicationSystem with component Service on multiple hosts) and distributed access (a Service with access points hosted on other systems).
`ServiceSAPDependency`	An association between a Service and a ServiceAccessPoint indicating that the referenced SAP is required for the Service to provide its functionality.
`SAPSAPDependency`	An association between a SAP and another SAP indicating that the latter is required in order for the former to utilize or connect with its Service.
`ServiceAccessBySAP`	An association that identifies the access points for a Service. For example, a printer may be accessed by Netware, Apple Macintosh, or Windows ServiceAccessPoints, potentially hosted on different Systems.

Example of an Extension into the Core Model

It is possible to develop many extensions into the Core Model. One possible extension includes the addition of a Managed Element class as an abstraction of the Managed System Element class. Descendents of this Managed Element class—classes that represent objects outside the managed system domain, such as Users or Administrators—may be added to the Core Model.

Common Model Schemas

The Common Model provides a set of base classes for the following technology-specific schemas.

Systems

The Systems Model describes the computer, application, and network systems that comprise the top-level system objects that make up the managed environment.

Devices

The Devices Model is a representation of the discrete logical units on the system that provide the basic capabilities of the system, such as storage, processing, communication, and input/output functions. There is a strong temptation to identify the system devices with the physical components of the system. This approach is incorrect because what is being managed is not the physical components themselves but rather the operating system's representation of the devices.

The representation provided by the operating system does not have a one-to-one correspondence with the physical components of the system. For example, a modem may correspond to a discrete physical component. It may just as well be provided by a multi-function card that supports a LAN adapter as well as a modem, or the modem may be provided by an ordinary process running on the system. It is very important in using or making extensions to the model to understand this distinction between Logical Devices and Physical Components and not to get them confused.

Applications

The CIM Application Management Model is an information model designed to describe a set of details that is commonly required to manage software products and applications. This model can be used for various application structures, ranging from

stand-alone desktop applications to a sophisticated, multiplatform, distributed, Internet-based application. Likewise, the model can be used to describe a single software product as well as a group of interdependent applications that form a business system.

A fundamental characteristic of the application model is the idea of the application life cycle. An application may be in one of four states: Deployable, Installable, Executable, and Executing. The interpretation and characteristics of the various objects used to represent applications are largely tied to the mechanisms used to transform applications from one state to another.

Networks

The Networks Model represents the various aspects of the network environment. This includes the topology of the network, the connectivity of the network, and the various protocols and services necessary to drive and provide access to the network.

Physical

The Physical Model provides a representation of the actual physical environment. Most of the managed environment is represented by logical objects, that is, objects that represent informational aspects of the environment rather than actual physical objects. Most of systems management is concerned with manipulating information that represents and controls the state of the system. Any impact on the actual physical environment (such as the movement of a read head on a physical drive, or the starting of a fan) is likely to only happen as an indirect consequence of the manipulation of the logical environment. As such, the physical environment is typically not of direct concern.

Apart from anything else, physical parts of the system are not instrumented. Their current state (and possibly even their very existence) can only be indirectly inferred from other information about the system. In the CIM, the physical model is a representation of this aspect of the environment and it is expected that it will differ dramatically from system to system and over time as technology evolves. It is also expected that the physical environment will always be very difficult to track and instrument, spawning the opportunity for a separate specialty, that of deploying applications, tools, and environments specifically aimed at providing information about the physical aspect of the managed environment.

Glossary

This Glossary defines terms used in the Sun WBEM documentation. Many of these terms are familiar to developers, but have new or altered meaning in the WBEM environment.

alias A symbolic reference in either a class or instance declaration to an object located elsewhere in a MOF file. Alias names follow the same rules as instance and class names. Aliases are typically used as shortcuts to lengthy paths.

aggregation relationship A relationship in which one entity is made up of the aggregation of some number of other entities.

association class A class that describes a relationship between two classes or between instances of two classes. The properties of an association class include pointers, or references, to the two classes or instances. All WBEM classes can be included in one or more associations.

Backus-Naur Form (BNF) A metalanguage that specifies the syntax of programming languages.

cardinality The number of values that may apply to an attribute for a given entity.

class A collection or set of objects that have similar properties and fulfill similar purposes.

CIM Object Manager Repository A central storage area managed by the Common Information Model Object Manager (CIM Object Manager). This repository contains the definitions of classes and instances that represent managed objects and the relationships among them.

CIM Schema A collection of class definitions used to represent managed objects that occur in every management environment.

See also core model, common model, and extension schema.

The CIM is divided into the metamodel and the standard schema. The metamodel describes what types of entities make up the schema. It also defines how these entities can be combined into objects that represent managed objects.

common model The second layer of the CIM schema, which includes a series of domain-specific but platform-independent classes. The domains are systems, networks, applications, and other management-related data. The common model is derived from the core model.

See also extension schema.

core model The first layer of the CIM schema, which includes the top-level classes and their properties and associations. The core model is both domain- and platform-independent.

See also common model and extension schema.

Distributed Management Task Force (DMTF) An industry-wide consortium committed to making personal computers easier to use, understand, configure, and manage.

domain The class to which a property or method belongs. For example, if status is a property of Logical Device, it is said to belong to the Logical Device domain.

dynamic class A class whose definition is supplied by a provider at runtime as needed. Dynamic classes are used to represent provider-specific managed objects and are not stored permanently in the CIM Object Manager Repository. Instead, the provider responsible for a dynamic class stores information about its location. When an application requests a dynamic class, the CIM Object Manager locates the provider and forwards the request. Dynamic classes support only dynamic instances.

dynamic instances An instance that is supplied by a provider when the need arises and is not stored in the CIM Object Manager Repository. Dynamic instances can be provided for either static or dynamic classes. Supporting instances of a class dynamically allows a provider to always supply up-to-the-minute property values.

enumeration Java term for getting a list of objects. Java provides an `Enumeration` interface that has methods for enumerating a list of objects. An individual object on this list to be enumerated is called an element.

extension schema The third layer of the CIM Schema, which includes platform-specific extensions of the CIM Schema such as Solaris and UNIX.

See also common model and core model.

flavor *See* qualifier flavor.

indication An operation executed as a result of some action such as the creation, modification, or deletion of an instance, access to an instance, or modification or access to a property. Indications can also result from the passage of a specified period of time. An indication typically results in an event.

inheritance The relationship that describes how classes and instances are derived from parent classes or superclasses. A class can spawn a new subclass, also called a child class. A subclass contains all the methods and properties of its parent class. Inheritance is one of the features that allows WBEM classes to function as templates for actual managed objects in the WBEM environment.

instance A representation of a managed object that belongs to a particular class, or a particular occurrence of an event. Instances contain actual data.

instance provider A type of provider that supports instances of system- and property-specific classes. Instance providers can support data retrieval, modification, deletion, and enumeration. Instance providers can also invoke methods.

See also property provider.

interface class The class used to access a set of objects. The interface class can be an abstract class representing the scope of an enumeration.

See also enumeration and scope.

Interface Definition Language (IDL) A generic term for a language that lets a program or object written in one language communicate with another program written in an unknown language.

key A property that is used to provide a unique identifier for an instance of a class. Key properties are marked with the Key qualifier.

Key qualifier A qualifier that must be attached to every property in a class that serves as part of the key for that class.

managed object A hardware or software component that is represented as a WBEM class. Information about managed objects is supplied by data and event providers as well as the CIM Object Manager Repository.

Managed Object Format (MOF) A compiled language for defining classes and instances. The MOF compiler (`mofc`) compiles `.mof` text files into Java classes and adds the data to the CIM Object Manager Repository. MOF eliminates the need to write code, thus providing a simple and fast technique for modifying the CIM Object Manager Repository.

management application An application or service that uses information originating from one or more managed objects in a managed environment. Management applications retrieve this information through calls to the CIM Object Manager API from the CIM Object Manager and from providers.

management information base A database of managed objects.

metamodel A CIM component that describes the entities and relationships representing managed objects. For example, classes, instances, and associations are included in the metamodel.

metaschema A formal definition of the Common Information Model, which defines the terms used to express the model, its usage, and its semantics.

method A function describing the behavior of a class. Including a method in a class does not guarantee an implementation of the method.

MOF file A text file that contains definitions of classes and instances using the Managed Object Format (MOF) language.

Named Element An entity that can be expressed as an object in the metaschema.

namespace A directory-like structure that can contain classes, instances, and other namespaces.

object path A formatted string used to access namespaces, classes, and instances. Each object on the system has a unique path which identifies it locally or over the network. Object paths are conceptually similar to Universal Resource Locators (URLs).

override Indicates that the property, method, or reference in the derived class overrides the similar construct in the parent class in the inheritance tree or in the specified parent class.

polymorphism The ability to alter methods and properties in a derived class without changing their names or altering interfaces. For example, a subclass can redefine the implementation of a method or property inherited from its superclass. The property or method is thereby redefined even if the superclass is used as the interface class.

Thus, the LogicalDevice class can define the variable status as a string, and can return the values "on" or "off." The Modem subclass of LogicalDevice can redefine (override) status by returning "on," "off," and "connected." If all LogicalDevices are enumerated, any LogicalDevice that happens to be a modem can return the value "connected" for the status property.

property A value used to characterize the instances of a class. Property names cannot begin with a digit and cannot contain white space. Property values must have a valid Managed Object Format (MOF) data type.

property provider A program that communicates with managed objects to access data and event notifications from a variety of sources, such as the Solaris operating environment or a Simple Network Management Protocol (SNMP) SNMP device. Providers forward this information to the CIM Object Manager for integration and interpretation.

qualifier A modifier containing information that describes a class, an instance, a property, a method, or a parameter. The three categories of qualifiers are: those defined by the Common Information Model (CIM), those defined by WBEM (standard qualifiers), and those defined by developers. Standard qualifiers are attached automatically by the CIM Object Manager.

qualifier flavor An attribute of a CIM qualifier that governs the use of a qualifier. WBEM flavors describe rules that specify whether a qualifier can be propagated to derived classes and instances and whether or not a derived class or instance can override the qualifier's original value.

range A class that is referenced by a reference property.

reference A special string property type that is marked with the reference qualifier, indicating that it is a pointer to other instances.

required property A property that must have a value.

schema A collection of class definitions that describe managed objects in a particular environment.

scope An attribute of a CIM qualifier that indicates which CIM elements can use the qualifier. Scope can only be defined in the Qualifier Type declaration; it cannot be changed in a qualifier.

selective inheritance The ability of a descendant class to drop or override the properties of an ancestral class.

Simple Network Management Protocol (SNMP) A protocol of the Internet reference model used for network management.

singleton class A WBEM class that supports only a single instance.

Solaris Schema A Sun extension to the CIM Schema that contains definitions of classes and instances to represent managed objects that exist in a typical Solaris operating environment.

standard schema A common conceptual framework for organizing and relating the various classes representing the current operational state of a system, network, or application. The standard schema is defined by the Distributed Management Task Force (DMTF) in the Common Information Model (CIM).

static class A WBEM class whose definition is persistent. The definition is stored in the CIM Object Manager Repository until it is explicitly deleted. The CIM Object Manager can provide definitions of static classes without the help of a provider. Static classes can support either static or dynamic instances.

static instance An instance that is persistently stored in the CIM Object Manager Repository.

subclass A class that is derived from a superclass. The subclass inherits all features of its superclass, but can add new features or redefine existing ones.

subschema A part of a schema owned by a particular organization. The Win32 and Solaris Schemas are examples of subschemas.

superclass The class from which a subclass inherits.

transitive dependency In a relation having at least three attributes R (A, B, C), the situation in which A determines B, B determines C, but B does not determine A.

trigger	A recognition of a state change (such as create, delete, update, or access) of a class instance, and update or access of a property. The WBEM implementation does not have an explicit object representing a trigger. Triggers are implied either by the operations on basic objects of the system (create, delete, and modify on classes, instances and namespaces) or by events in the managed environment.
Unified Modeling Language (UML)	A notation language used to express a software system using boxes and lines to represent objects and relationships.
Unicode	A 16-bit character set capable of encoding all known characters and used as a worldwide character-encoding standard.
UTF-8	An 8-bit transformation format that may also serve as a transformation format for Unicode character data.
virtual function table (VTBL)	A table of function pointers, such as an implementation of a class. The pointers in the VTBL point to the members of the interfaces that an object supports.
Win32 Schema	A Microsoft extension to the CIM Schema that contains definitions of classes and instances to represent managed objects that exist in a typical Win32 environment.

Index

D

E

H

I

J

L

M

N

O

P

Q

R

S

T

U

W